Digitale Erweiterung eines Brettspiels

Digitale Erweiterung eines Brettspiels

Masterarbeit

Thomas Lohmüller
<lthomas@ethz.ch>

Betreuer
Steve Hinske

Prof. Friedemann Mattern
Distributed Systems Group
Institute of Parvasive Computing
Department of Computer Science
ETH Zürich

28. Oktober 2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Brettspiele erfreuen sich, trotz starker Konkurrenz durch Computerspiele, einer grossen Beliebtheit. Speziell der soziale Aspekt bei Spielen wie Warhammer 40'000 wird von den Spielern geschätzt. Das Ziel dieser Arbeit war die Erweiterung eines solchen Spielfelds mittels RFID-Technologie. Eine unter dem Feld angebrachte bewegliche RFID-Antenne wird benutzt, um die Position und Ausrichtung der Spielfiguren, welche mit RFID-Tags markiert werden, zu erkennen. Der erstellte Prototyp hat gezeigt, dass eine auf wenige Millimeter genaue Erkennung möglich ist.

Des weiteren wurde ein Programm geschrieben, das die erfassten Daten grafisch darstellen und anhand der Regeln auf Gültigkeit prüfen kann. Die grundlegendsten Regeln, ohne die Warhammer 40'000 nicht funktioniert, wurden integriert. So entstand ein virtuelles Ebenbild des physikalischen Spielfelds, das nicht nur ungültige Züge erkennen kann, sondern den Spielern auch jederzeit das Beziehen von Informationen zum aktuellen Spielstand erlaubt.

Diese Arbeit hat gezeigt, dass mittels RFID-Technologie selbst für solche Anforderungen genügend genaue Positionsangaben erfasst werden können. Auch zeigt sich das Potential solcher Produkte, die den Spielern das mühselige Messen von Distanzen und das Nachschlagen der Attribute von Spielfiguren abnehmen. In weiteren Arbeiten könnte aus diesem Prototypen ein interessantes Produkt entstehen.

Inhaltsverzeichnis

Die Abenteuer von heute finden in den Computern und den Laboratorien statt.

Thornton Wilder

1 Einleitung

Nebst den unzähligen neuen Computer- und Konsolenspielen haben auch die traditionellen Brettspiele nie ihren Reiz verloren und erfreuen sich grosser Beliebtheit. Denn diese Spiele bieten einiges, das auch modernste Online-Spiele nicht oder zumindest nicht in dieser Art bieten können.

Bei vielen dieser Spiele können Figuren und Umgebung selbst gestaltet werden. Dabei sind der Kreativität keine Grenzen gesetzt, und schöne Modelle und Spielfelder wechseln auch öfters in Online-Auktionen für viel Geld ihren Besitzer. Dazu kommt der Aspekt des Sammelns und Tauschen von Modellen unter Freunden. Viele Erwachsene arbeiten während des Tages an einem Computer und bevorzugen auch aus diesem Grund ein Hobby, das nicht mit einem Computer zusammenhängt. Die Spieler holen Spielfeld und Figuren, stellen die Armeen bei einem Glas Wein auf und starten mit dem Spiel, dabei über verschiedenste Themen diskutierend.

Das Fehlen ausgeklügelter Sound- und Grafikeffekten mag auf den ersten Blick eher als Nachteil der Brettspiele wirken, wird aber gerade von älteren Spielern sehr geschätzt. Der Fokus liegt auf der Taktik und nicht auf einzelnen Effekten oder Figuren. Speziell bei Miniatur-Kriegsspielen ist die Komplexität der Regelwerke oft sehr hoch, und die Möglichkeiten sind beinahe unbegrenzt. Bei Unklarheiten wird das in der Gruppe diskutiert und eventuell im Regelwerk nachgeschlagen. Im Gegensatz zu Computer- und Konsolenspielen ist der soziale Aspekt bei diesen Brettspielen sehr wichtig, und da die meisten Spiele auf einzelnen Runden basieren, lässt das Spiel auch genügend Zeit für Gespräche.

1.1. Motivation

Neben diesen von den Spielern sehr geschätzten Eigenschaften gibt es jedoch durchaus Potenzial für Verbesserungen. So hat sich im Gespräch mit Spielern gezeigt, wo Änderungen erwünscht sind.

Messen: Diese Spiele haben ein Spielfeld, auf dem die Figuren frei positioniert werden können. Oft muss der Abstand zwischen zwei Figuren gemessen oder die zurückgelegte Distanz bei einer Bewegung kontrolliert werden.

Nachschlagen: Die einzelnen Figuren besitzen verschiedenste Attribute, die durch das Spiel immer wieder benötigt werden. Nur wenige Spieler kennen alle Werte ihrer Modelle auswendig, Daher müssen häufig Eigenschaften nachgeschlagen werden.

Vereinfachung: Grössere Mengen von Figuren sind oft unübersichtlich und nicht einfach zählbar. Das selbe Problem gibt es bei vielen Würfeln, bei denen auch das Resultat überprüft werden muss.

Archivierung: Im Internet findet man von Spielern angelegte Seiten, die den kompletten Spielablauf zu dokumentieren versuchen. Dazu werden Fotos jeder Runde und Texte benutzt. Das Dokumentieren ist einerseits bei Experimenten mit neuen Einheiten, aber auch bei Turnieren interessant, um den Verlauf zu dokumentieren.

Viele dieser Punkte bezeichnen Spieleigenschaften, die bei vielen Computerspielen selbstverständlich sind. Niemand misst die Distanz auf dem Computerbildschirm, auch die Eigenschaften der Einheiten müssen nicht mühsam in einem Buch nachgeschlagen werden. Viele Computerspiele besitzen auch eine Replay-Funktion, mit der das ganze Spiel wiederholt werden kann.

Das Ziel ist daher die Kombination der Vorteile von Brettspielen mit denjenigen von Computerspielen. Eine der wichtigsten Aufgaben ist dabei das Erfassen der Figuren auf dem Spielfeld um für den Computer nutzbare Daten zu gewinnen.

1.2. Ziel dieser Arbeit

Auf der Basis der Resultate von bisherigen Arbeiten [8, 17, 18] sollte der Prototyp eines Spielfelds für Brettspiele erstellt werden. Dabei werden nicht wie bei obigen Arbeiten mehrere unabhängige RFID-Antennen genutzt, sondern nur eine einzige, die unter dem Spielfeld bewegt wird. Durch diese Bewegung soll versucht werden, die Auflösung beim Erkennen der Objekte soweit zu verbessern, dass diese in einem Brettspiel genutzt werden kann.

Das Ziel ist eine Umsetzung mit dem Spiel Warhammer 40'000 der Firma Games Workshop. Das Spiel zeichnet sich durch ein grosses Spielfeld aus, auf dem die Figuren frei positioniert werden können, und besitzt ein sehr vielfältiges Regelwerk. Es gilt, ein Programm zu entwickeln, das den Spieler unterstützt, jedoch möglichst wenig ins Spielgeschehen eingreift. Nebst der Verwaltung der Zustände aller Figuren gehört zu den Funktionen des Programms

auch das Messen von Distanzen und die Entscheidung, ob sich zwei Figuren sehen können oder nicht.

Die Anwendung soll einerseits als Basis für weitere Projekte genutzt werden können, sollte andererseits aber auch bereits genug Funktionen haben, um einen vollständigen Spielablauf zu ermöglichen. Einzelne Einheiten und Fahrzeuge sollten grundsätzlich integriert werden, die Regeln müssen jedoch nicht komplett abgedeckt werden.

1.3. Aufbau dieses Dokuments

Einige Begriffe wie zum Beispiel *Tangible User Interface* wurden nicht übersetzt, da diese Begriffe etabliert sind und auch keine passende deutsche Übersetzung existiert. Auch andere in der Informatik gebräuchliche Begriffe wurden nicht vollständig übersetzt. Klassendiagramme, Grafiken der Architektur und auch Kommentare im Quelltext sind in Englisch verfasst, da diese Bestandteile der vollständig in Englisch gehaltenen Software-Dokumentation sind.

In dieser Arbeit wird im Kontext des Spiels unter einer *Einheit* immer eine Anzahl zusammengehöriger Modelle verstanden. Diese *Modelle*, oder auch *Figuren* bezeichnen die Miniatur-Figuren auf dem Spielfeld und es wird zwischen *Fahrzeugen* und *einfachen Modellen* unterschieden.

Nach dieser kurzen Einführung in die Thematik wird im Kapitel *Hintergrund* (ab Seite 5) das Umfeld beschrieben. Nebst Verweisen zu verwandten Arbeiten enthält dieses Kapitel auch eine kurze Einführung in Warhammer 40'000, RFID und andere verwandte Themen.

Im Kapitel *Konzept* (ab Seite 13) wird auf Basis der durch das Spiel definierten Anforderungen ein Konzept für die Umsetzung erarbeitet und auch eine grundlegende Architektur für die Anwendung festgelegt.

Nach dem Konzept folgt die *Umsetzung* (ab Seite 29). In diesem Kapitel wird detailliert auf das Vorgehen bei der Umsetzung, auf Probleme und deren Lösungen eingegangen.

Es folgt das *Fazit* (ab Seite 67). Nach einer kurzen Wiederholung der wichtigsten Erkenntnisse aus Konzept und Umsetzung wird auf mögliche zukünftige Projekte und auch die gemachten Erfahrungen bei diesem Projekt eingegangen.

Im Anhang befinden sich Hinweise zur Inbetriebnahme und Bedienung der für diese Arbeit geschriebenen Software, das Literaturverzeichnis und die eidesstattliche Erklärung.

Sämtliche Fotos, Grafiken, Auswertungen und die erstellten Programme aus dieser Arbeit sind auch in digitaler Form verfügbar. Der Quelltext ist dokumentiert und kann jeweils als weitere Informationsquelle beigezogen werden.

Sicher können Computer Probleme lösen, Informationen speichern, kombinieren und Spiele spielen – aber es macht ihnen keinen Spass.

Leo Rosten

2 Hintergrund

Seit Jahren wird versucht, Brettspiele mit verschiedensten Technologien anzureichern, um das Spielerlebnis zu verbessern. Die unregelmässig stattfindende *PerGames*-Konferenz [5] bietet einen guten Überblick über die aktuellen Projekte. Von kleinen mit RFID versehenen Würfeln [19] bis zu stadtweiten oder sogar weltweiten Spielen ist alles dabei.

Aufgrund der Breite dieses Gebiets wird in diesem Dokument nur sehr spezifisch auf ähnliche Arbeiten eingegangen. Bereits in der Aufgabenstellung war festgelegt, dass Warhammer 40'000 mit einem RFID-Spielfeld erweitert werden soll. Dieses Spiel und die RFID-Technologie haben die Planung daher dominiert.

2.1. Warhammer 40'000

Warhammer 40'000 [6] (WH40k) gehört zu den Miniatur-Kriegsspielen und spielt im 41. Jahrtausend. Die Menscheit ist auf vielen Planenten verteilt und kämpft gegen zahlreiche andere Spezien ums Überleben. Das Spiel wurde 1987 von der Firma Games Workshop [1] zum ersten Mal veröffentlicht und wird seither regelmässig überarbeitet.

Ein einzelnes Spiel, Szenario genannt, stellt jeweils einen Konflikt zwischen zwei oder mehreren Parteien dar. Jede Partei hat auf dem Spielfeld verschiedenste Einheiten, die wiederum aus mehreren Modellen bestehen. Die Fähigkeiten jeder Figur sind in den Regeln festgelegt, ebenso der *Wert* der Einheit. Starke Einheiten haben einen höheren Wert, schwächere Einheiten einen niedrigeren. Vor dem Spiel einigen sich die Mitspieler jeweils auf einen Wert, den die einzelnen Armeen nicht überschreiten dürfen. Die Werte einzelner Figuren lassen

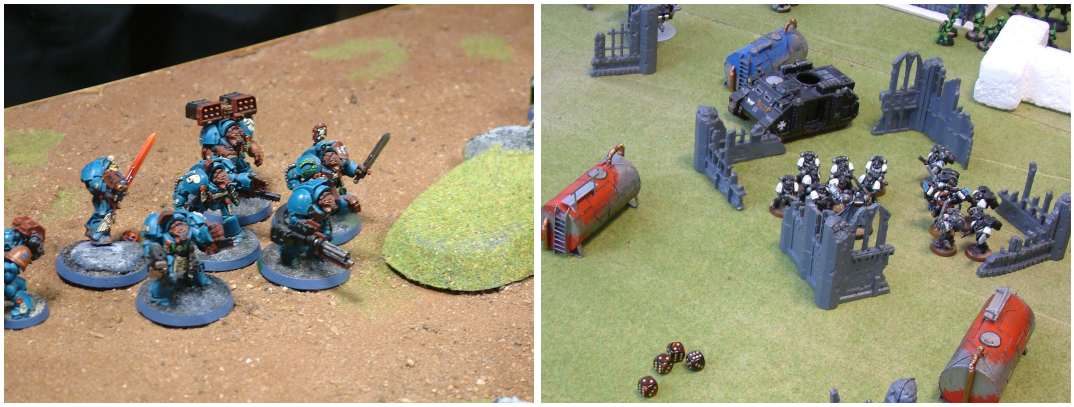


Abbildung 2.1: Auschnitte aus einem WH40k Spiel.

sich durch verschiedene Ausrüstungsgegenstände noch verändern. Welche Modelle welche Ausrüstung tragen, muss den Mitspielern vor Spielbeginn kommuniziert werden.

Nebst der maximalen Punktezahl für eine Armee gibt es viele zusätzliche Regeln, die eingehalten werden müssen, um eine gültige Armee aufzustellen. Die Beschreibung der Regeln teilt sich auf ein allgemein gültiges Regelwerk [6] sowie den sogenannten *Codex*, der jeweils für ein bestimmtes Volk alle Einheiten, Ausrüstungsgegenstände und einige Hintergrundinformationen enthält. in der Praxis wird häufig ArmyBuilder [12] genutzt, um unkompliziert eine allen Regeln entsprechende Armee zusammenzustellen.

Auch das Spielfeld kann sehr vielfältig aufgebaut sein und wird vor Spielbeginn mit allen Parteien besprochen. Sind die Vorbereitungen abgeschlossen, beginnt das eigentliche Spiel, das auf einzelnen Runden aufgebaut ist. In jeder Runde ist ein Spieler am Zug und führt die Spielphasen *Bewegen*, *Schiessen* und *Nahkampf* der Reihe nach aus. Für die meisten Aktionen müssen Würfel geworfen werden. Der Umgang mit vielen Würfeln ist dabei typisch für dieses Spiel und macht, nach Aussagen langjähriger Spieler, auch einen Teil des Reizes aus.

Das Szenario kann auf sehr vielfältige Weise enden. Wie das genau abläuft, muss von den Mitspielern vor Spielbeginn festgelegt werden. Einmal muss ein Spieler nur eine vorgegebene Anzahl Runden überleben oder vielleicht ein spezielles Ziel mit einer Einheit erreichen, in anderen Szenarien müssen die Gegner vernichtet werden.

Das Spielfeld, Gebäude, Gelände und natürlich auch die einzelnen Figuren können von den Spielern selbst bemalt oder auch selbst gebaut werden. Häufig werden Standardfiguren mit persönlichen Gegenständen erweitert. Vielen Spielern sind die Figuren und Geländemodelle so wichtig, dass diese einzeln in mit Schaumstoff gefütterten Koffern transportiert werden. Die meisten Modelle sind aus Plastik, wertvollere oft auch aus Metall.

Games Workshop hat zwei verwandte Spiele im Angebot, die beide im selben Umfeld wie WH40k spielen. Dies ist einerseits *Raumflotte Gothic* [31], wo mit grossen Flotten im Weltraum gekämpft wird, und das Spiel *Necromunda Unterwelt* [30], das Strassenschlachten zwischen kleinen Gruppen simuliert. Das Spielprinzip ist jeweils ähnlich.

2.2. Game Programming

Aus dem Klassiker *The Art of Computer Game Design* [10], aber auch aus *Killer Game Programming with Java* [11] wurden einige Ideen für die Umsetzung der Spiellogik und der Architektur bezogen. Das erste Buch [10, Seite 39] enthält einen Absatz, der Spiele wie WH40k gut beschreibt:

> Wargames are easily the most complex and demanding of all games available to the public. Their rules books read like contracts for corporate mergers and their playing times often exceed three hours. Wargames have therefore proven to be very difficult to implement on the computer; we have, nevertheless, seen entries.

Dieses Projekt weicht in vielen Punkten von klassischer Spielprogrammierung ab. So spielt die Geschwindigkeit kaum eine Rolle, da einzelne Runden meistens über 15 Minuten dauern. Auch müssen für die Spieloberfläche keine aufwändigen grafischen Effekte erstellt werden, da diese Oberfläche das Spielgeschehen so wenig als möglich beeinflussen sollte. Das zu implementierende Programm muss auch keinen Gegner steuern und taktische Entscheidungen treffen können. Es beschränkt sich auf eine Überwachung.

2.3. Augmented Tabletop Games

Die Erweiterung von Brettspielen mit zusätzlicher Technologie erfolgt meistens auf die selbe Art. Es werden Möglichkeiten gesucht, wie der Computer möglichst effizient die Stellung der Spielfiguren mitverfolgen kann, um darauf irgendwie reagieren zu können. Je nach Spiel ist dazu eine mehr oder weniger genaue Lokalisierung erforderlich.

Grob lassen sich die Spiele in zwei Arten unterteilen (Abbildung 2.2). Die meisten klassischen Brettspiele wie Mühle oder Schach gehören zu der ersten Art, da sie eine Anzahl verschiedener Felder haben, auf denen sich die Figuren befinden können. Auch einige Kriegsspiele, wie beispielsweise *Classic BattleTech* [22], haben einen vorgegebenen Raster, auf dem die Figuren bewegt werden können. Für so ein System wurde mit *Wizard's Apprentice* [24] ein Spielfeld entwickelt, das mit RFID arbeitet. Dazu wurde unter jedem möglichen Feld eine Antenne montiert, und diese werden der Reihe nach abgefragt. Um nicht alle Antennen bei jedem Durchgang abfragen zu müssen, wurde zusätzlich ein Helligkeitssensor unter jedem Feld montiert. Erst wenn dieser durch eine Figur abgedeckt wird, wird die RFID-Antenne abgefragt. Solch ein Vorgehen ist ideal, wenn es nur eine relativ kleine Anzahl möglicher Positionen gibt.

Bei der anderen Art von Spielen können die Figuren absolut frei positioniert werden. Je nach Spiel zählt jeder Zentimeter oder sogar jeder Millimeter. Die maximale Bewegung der Figuren wird dann nicht in der Anzahl Felder, sondern in einer Längenangabe wie zum Beispiel 6 Zoll angegeben. Diese Spiele führen zu ganz anderen Anforderungen an die Technik. Denn es müssen Lokalisierungsverfahren wie die Triangulation herangezogen werden, was einen wesentlich grösseren Aufwand bedeutet. Die Verfahren werden im nächsten Abschnitt näher erläutert.

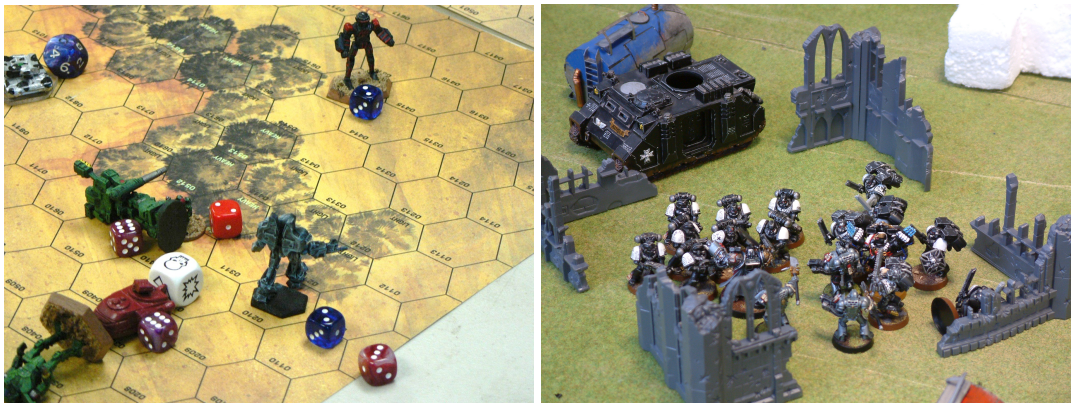


Abbildung 2.2: Classic BattleTech (links), WH40k (rechts).

Eine andere Art der Augmented Spiele benutzt Spielkarten, die die Einheiten und deren Fähigkeiten repräsentieren. Bei dieser Art Spiel ist es meistens nur wichtig, die Karten korrekt und schnell zu erkennen. Die Position ist oft nebensächlich oder auf sehr wenige Bereiche begrenzt.

2.4. Position und Ausrichtung von Objekten erkennen

Es gibt verschiedenste Möglichkeiten, Objekte zu lokalisieren und auch deren Ausrichtung zu bestimmen. Von Ultraschall [25] bis zu Infrarot [21] wurde mit alle Varianten experimentiert. Dabei lassen sich die Systeme nach Hightower [15] in die Kategorien *Triangulation*, *Scene Analysis* und *Proximity* einteilen.

2.4.1. Klassifikation

Bei *Triangulation* wird die Position eines Objekts über die Eigenschaften von Dreiecken berechnet. Dies ist über Winkel oder über Distanzangaben machbar. Typische Beispiele sind GPS (Berechnung über Distanz) oder die im Flugsektor verwendeten Drehfunkfeuer (VOR, Berechnung über Winkel). Beide Systeme erlauben dem Objekt, anhand bekannter Position der Referenzobjekte seine eigene Position festzustellen. Es ist aber auch für die Umwelt möglich, die Position eines Objekts zu bestimmen. Dies wird beispielsweise bei Positionsbestimmungen im GSM und UMTS-Netz genutzt [14]. Ein System für den Innenbereich ist SpotON [16] und erreicht eine Auflösung von ca. $1m^3$.

Die *Scene Analysis* ist nur vom Objekt selbst aus möglich. Dabei wird die Umgebung beobachtet und anhand gefundener Informationen, wie zum Beispiel dem Horizont, und einer vorbereiteten Datenbank versucht, die eigene Position zu bestimmen. Auch Variationen ohne vorbereitete Datenbank sind möglich, können jedoch nur die relative Positionsveränderung feststellen [26]. Das RADAR-Projekt [9] nutzt dieses Verfahren zusammen mit WLAN-Funksignalen.

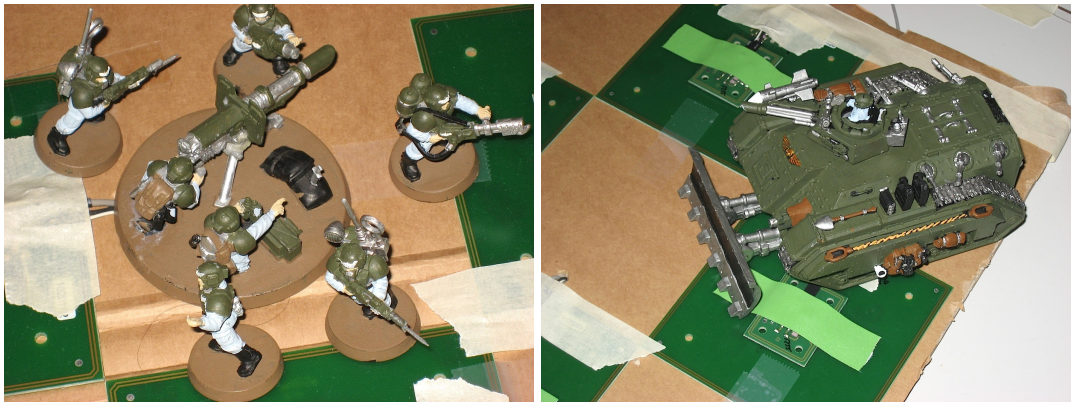


Abbildung 2.3: Lokalisierung mittels RFID aus [8].

Mit *Proximity* sind Verfahren gemeint, bei denen festgestellt wird, ob sich ein Objekt in der Nähe eines bekannten Punkts befindet. Ist die Position eines WLAN-Netzes bekannt und wird ein Funksignal von diesem Netz empfangen, so muss sich der Empfänger in der Nähe befinden. In diese Kategorie gehört auch die RFID-Technologie. Sieht ein Leser einen Tag, so weiss das System, dass der Tag in der Nähe der Antenne sein muss.

2.4.2. Verwendung in Brettspielen

In *Enhancing Tabletop Games with Relative Positioning* [21] werden Prototypen für aktive Modelle vorgestellt, die über Ultraschall oder Infrarot die örtliche Ausrichtung der Figuren untereinander bestimmen können. Das System weiss, unter welchem Winkel mit welcher Distanz andere Figuren stehen. Der Vorteil dieser Variante ist, dass keine Änderungen am Spielfeld vorgenommen werden müssen. Der Nachteil ist die teure Technik, die in jede Figur integriert werden muss. Dies ist bei WH40k aufgrund der grossen Anzahl Figuren kaum realisierbar. Ausserdem wird die Auflösung mit „einigen Zentimetern" angegeben, was für WH40k nicht ausreicht. Dieses System basiert auf Triangulation. Es werden Winkel gemessen und Signallaufzeiten bestimmt.

Auch mit günstigen, passiven RFID-Tags wurde bereits gearbeitet [18, 8]. Man hat RFID-Antennen schachbrettartig auf dem Spielfeld verteilt und über einen Multiplexer nacheinander abgefragt (Abbildung 2.3). Da die Antennen nicht in der Lage sind, Einfallswinkel oder Signalstärke der erhaltenen Signale zu messen, reduzierte sich die Auflösung auf einige Zentimeter. Das System basiert auf dem Prinzip *Proximity* oder auch *Cell of Origin*.

Das für diese Arbeit geplante Verfahren mit einer RFID-Antenne, die unter dem Spielfeld bewegt wird, lässt sich nicht einfach in eine der obigen Kategorien einteilen. Es ist eine Kombination aus einer auf Distanzmessung basierenden Triangulation (die Bewegung der Antenne) und dem Prinzip Proximity zum Erkennen von RFID-Tags in Reichweite. Durch das Verschieben der Antenne werden sehr viele örtlich nahe beieinander liegene Antennen simuliert. Man könnte es auch als *Cell of Origin* mit sehr vielen, jedoch nicht gleichzeitig existierenden Zellen beschreiben.

2.5. Tangible User Interface

Ein *Tangible User Interface* [28] (Tabgible UI) zeichnet sich durch folgende Charakteristiken aus:

1. Physikalische Objekte sind mit digitalen Informationen verbunden.
2. Physikalische Objekte enthalten Mechanismen zur gegenseitigen Kontrolle.
3. Physikalische Objekte werden, für den Anwender wahrnehmbar, mit digitalen Objekten verbunden.
4. Der Zustand physikalischer Objekte zeigt den Zustand des digitalen Systems.

Obige Punkte und Arbeiten wie beispielsweise [20] zeigen, dass das Ziel dieser Arbeit kein vollständiges Tangible UI darstellt. Die Verbindung digitaler und physikalischer Objekte findet nur in eine Richtung statt. Punkt 1 und 3 sind erfüllt. Mit den Figuren auf dem Spielfeld werden zusätzliche Informationen verbunden, und durch die grafische Repräsentation des Spielfelds auf dem PC ist für den Anwender klar, dass diese Objekte, wenn auch nur einseitig, miteinander verbunden sind. Dem Computer fehlt jedoch die Möglichkeit, die physikalischen Figuren zu beeinflussen (Punkt 2), und er kann den aktuellen Zustand nur über den Bildschirm ausgeben, was Punkt 4 nicht erfüllt.

Weiter ist eine Interaktion mit dem PC bei diesem Prototypen noch notwendig. Es handelt sich daher um kein echtes Tangible User Interface. Zukünfige Arbeiten könnten dieses Projekt weiter in diese Richtung führen.

2.6. RFID-Technologie

Die *Radio Frequency Identification*-Technologie (RFID) wurde zum Erkennen der Anwesenheit von Transpondern (Tags) entwickelt. Eine gute Einführung dazu bietet *An Introduction to RFID Technology* [27]. Grob lassen sich aktive und passive Tags unterscheiden. Während die ersteren eine grössere Reichweite und höhere Leserate haben, benötigen die passiven Tags keine integrierte Energieversorgung und sind daher wartungsfrei.

Über RFID wurden sehr viele Arbeiten geschrieben. Da für dieses Projekt nur die Lokalisierung von RFID-Tags interessant ist, werden auch nur Arbeiten in diesem Umfeld vorgestellt.

Obwohl RFID ursprünglich entwickelt wurde, um die Anwesenheit von Objekten, nicht aber deren genaue Position oder Ausrichtung festzustellen, ist das bereits in einigen Arbeiten untersucht worden. Speziell für die Integration in Brettspiele eignet sich diese Technologie sehr gut.

- Die passiven RFID-Tags sind sehr klein und wartungsfrei. Sie können unsichtbar an den Spielfiguren angebracht werden und sollten über viele Jahre funktionieren. Defekte Tags können einfach ausgetauscht werden.
- Werden Referenz-Tags im Spielfeld integriert, wird keine Kalibrierung benötigt.

- Die Figuren können eindeutig identifiziert werden, auch wenn sehr viele ähnliche Figuren nahe beieinander stehen.
- Die Kosten für die Hardware sind im Vergleich zu anderen Technologien sehr gering.
- Die Figuren werden auch erkannt, wenn diese auf niedrigen Geländeelementen stehen. Ein direkter Kontakt wird nicht benötigt.

Bei anderen Technologien sind meistens die Tags zu gross, die Technik zu teuer oder die Objekte können nicht eindeutig identifiziert werden [18]. Die Bilderkennung verspricht eine sehr genaue Lokalisierung, benötigt jedoch sehr viel Rechenleistung und kann die beinahe identisch aussehenden Figuren nicht unterscheiden.

In *Utilizing RFID Signaling Scheme for Localization of Stationary Objects and Speed Estimation of Mobile Objects* [29] wurde eine Methode aufgezeigt, die Position eines Tags im eindimensionalen Raum zu bestimmen. Dies entspricht der Entfernung zwischen Tag und Leser. Durch Variieren der Signalstärke versuchte man die Distanz abzuschätzen, wobei jeweils die Anzahl erfolgreicher Lesevorgänge in einer gewissen Zeiteinheit ermittelt wurde. Dabei wurde mit Distanzen von wenigen Metern gearbeitet. Durch das Hinzufügen weiterer Antennen liesse sich dies auch auf den zwei- und dreidimensionalen Raum erweitern. Das System bietet jedoch bei weitem nicht die für dieses Projekt benötigte Präzision.

In LANDMARC [23] wurde ein Projekt zur Lokalisierung aktiver RFID-Tags in Gebäuden vorgestellt. Dabei wurde auch eine Variation der Signalstärke der Leser genutzt, um die Distanz abzuschätzen, und mehrere Leser ermöglichten eine ungefähre Abschätzung der Position. Installierte Referenz-Tags erhöhten die Präzision bis zu einer Auflösung von ungefähr einem Kubikmeter.

In den bereits erwähnten Arbeiten [18, 17, 8] wurde ein Array von RFID-Antennen dazu genutzt, eine Lokalisierung von Tags durchzuführen. Dies konnten bis auf wenige Zentimeter genau lokalisiert werden. Die Auflösung lässt sich jedoch kaum mehr erhöhen. Das elektromagnetische Feld der Antenne springt auf andere Antennen über und kann daher auch Tags erkennen, die weiter entfernt sind. Dies führt zu falschen Resultaten, was eine genaue Lokalisierung verunmöglicht.

Für diese Arbeit wird eine wesentlich feinere Auflösung angestrebt, und daher ist die geringe Reichweite der passiven RFID-Tags sogar von Vorteil. Anstatt die Anzahl Antennen unter dem Spielfeld weiter zu erhöhen, wird nur eine einzige bewegliche Antenne eingesetzt. Die Tags in Reichweite der Antenne werden abgefragt, die Antenne ein paar Millimeter bewegt und das Vorgehen wiederholt. Dies entspricht sehr vielen - sich gegenseitig überlappenden - Antennen unter dem Spielfeld, ohne dass das elektromagnetische Feld von Antenne zu Antenne springen kann.

Es ist offensichtlich, dass eine bewegte Antenne sehr viel mehr Zeit benötigt, um ein ganzes Spielfeld einzulesen als mehrere stationäre Antennen. Dies spielt aber bei Spielen wie WH40k kaum eine Rolle, da einzelne Runden meistens über 15 Minuten dauern.

2.7. LEGO Mindstorms

Sucht man einen Roboterbausatz für Prototypen, gibt es wenige Alternativen zu LEGO Mindstorms. Das aktuelle NXT-System bietet eine programmierbare Steuereinheit mit 32-Bit-Mikroprozessor, drei Ausgängen für Motoren und vier Sensoranschlüssen. Auf die Steuereinheit lassen sich Programme übertragen, die einem Roboter autonomes Handeln ermöglichen, oder man steuert die Einheit direkt von einem PC aus.

Die offizielle Programmierschnittstelle von LEGO ist ein grafisches, auf LabVIEW basierendes Produkt. Es gibt jedoch viele andere Möglichkeiten von BASIC über C, C++ bis hin zu Java. Auch für die Kommunikation mit dem PC gibt es verschiedene Varianten. Während das Vorgängermodell RCX ausschliesslich über Infrarot mit dem PC kommunizierte, bietet das NXT-System Bluetooth und USB-Kommunikation.

Ein weiterer Vorteil von LEGO Mindstorms sind die vielen vorhandenen LEGO-Teile und die Bekanntheit des Systems. So lassen sich einfache Roboter in kurzer Zeit erstellen und sehr vielfältig programmieren.

Leider vertreibt LEGO keinen Netzadapter zu diesem Set. So müssen ständig sechs Batterien in der Steuereinheit sein, die bei intensiver Nutzung in ca. 30 Minuten leer sind.

Man versteht etwas nicht wirklich, wenn man nicht versucht, es zu implementieren.

Donald Ervin Knuth

3 Konzept

Das Ziel dieser Arbeit ist das Erweitern des Brettspiels Warhammer 40'000 mittels RFID-Technologie. RFID soll genutzt werden, um die Position und Ausrichtung der Objekte auf dem Spielfeld zu bestimmen. Ein Computerprogramm soll diese Daten auswerten, grafisch darstellen und die Spieler beim Einhalten der Regeln unterstützen.

Dieses Kapitel beginnt mit den *Anforderungen*, die sich durch WH40k und RFID an dieses Projekt ergeben. Danach folgt eine Erläuterung der verschiedenen *Module* und einige *Ablaufdiagramme*, die zeigen, wie ein WH40k-Spiel funktioniert. Die letzten beiden Abschnitte befassen sich mit dem Ablauf dieses Projekts und geben einige Hinweise auf Schnittstellen für zukünftige Projekte.

3.1. Anforderungen

Das Programm soll während eines Spiels den Status auf dem Bildschirm anzeigen und anhand der Spielregeln feststellen, ob die Bewegungen der Modelle regelkonform sind.

Das WH40k-Regelwerk [6] enthält eine detaillierte Beschreibung aller Regeln. Für diese Arbeit sollen nur die grundlegendsten Regeln und wenige Einheiten implementiert werden. Die nachfolgende Tabelle 3.1 zeigt die wichtigsten Anforderungen auf, ersetzt aber nicht das Regelwerk. Die Ablaufdiagramme im Kapitel 3.3 zeigen detaillierter, wie das Spiel funktioniert.

Einige der Anforderungen, wie zum Beispiel 2.b, sind sehr ungenau definiert. Dies liegt daran, dass zu Projektbeginn die zu erreichenden Werte noch nicht bekannt waren.

Tabelle 3.1.: Anforderungen (Priorität: A(höchste) bis C, X ist eine Voraussetzung)

Nummer	Titel	Beschreibung	Prio
1	Hardware	LEGO Mindstorms und Feig RFID-Lesegerät sollen genutzt werden	X
2.a	Spielfeld	Die RFID-Antenne soll unter dem Spielfeld bewegt werden	X
2.b		Die Abweichung beim Abtasten des Spielfelds sollten höchstens $1cm$ betragen.	A
2.c		Kleine Modelle werden mit einem RFID-Tag markiert, grössere mit mindestens zwei	X
3	Anzahl Spieler	Es gibt mindestens zwei Spieler, keinen Computergegner	A
4	Spielphasen	Die Phasen *Bewegen*, *Schiessen* und *Nahkampf* sollten implementiert sein	X
4.1.a	Phase: Bewegen	Funktioniert grundsätzlich, testet auf Kohärenz der Einheit sowie zurückgelegtem Weg	A
4.1.b		Gefährliches und schwieriges Terrain wird beachtet	B
4.1.c		Überlappung mit anderen Modellen wird erkannt	B
4.1.d		Berechnen der Weglänge um Hindernisse herum anstatt nur Luftlinie	C
4.2.a	Phase: Schiessen	Funktioniert grundsätzlich, testet auf Distanz	A
4.2.b		Testen auf Sichtkontakt (keine Objekte im Weg)	B
4.2.c		Testen der Höhe der betroffenen Objekte	C
4.2.d		Beachten der Deckung	C
4.3.a	Phase: Nahkampf	Funktioniert grundsätzlich	A
4.3.b		Beachten der Deckung	C
5.a	GUI	Darstellen des Spielfelds und der Einheiten	A
5.b		Anzeigen des Zustands aller Objekte	B
5.c		Hinterlegen des Spielfelds mit einem Bild des echten Spielfelds	B
6	Konfiguration	Konfigurationsänderungen sollten möglich sein ohne Neukompilieren des Programms	A
7.a	Zukunft	Vollständige Dokumentation des Quelltexts	A
7.b		Vorbereiten des Programms für spätere Erweiterungen	B

3.2. Module

Um die Erweiterbarkeit zu garantieren, empfiehlt sich ein Aufteilen der Arbeit in verschiedene, möglichst unabhängige Module.

Abbildung 3.1: Übersicht über die geplanten Module.

Obige Abbildung 3.1 zeigt die geplanten Module, die in den nächsten Abschnitten noch detaillierter beschrieben werden. Die Komponenten lassen sich unabhängig voneinander erstellen und testen. Für komplexere Methoden sollen auch Unit-Tests geschrieben werden. Von der Logik her macht es Sinn, die Module in der Reihenfolge des Datenflusses (Quelle zur Senke) zu erstellen. Dieses Vorgehen soll während der Implementierung genutzt werden.

3.2.1. Bau der Hardware

Hierbei handelt es sich nicht um ein Modul im softwaretechnischen Sinn, aber um einen wichtigen Teil dieser Arbeit, weshalb das hier auch erwähnt werden soll.

Die RFID-Antenne wird mittels einer Konstruktion aus LEGO-Elementen sowie dem LEGO Mindstorms Robotik-Set unter dem Spielfeld bewegt und ständig abgefragt (Abbildung 3.2). Für diese Arbeit wird ein kleines Spielfeld erstellt, die genaue Grösse jedoch nicht festgelegt.

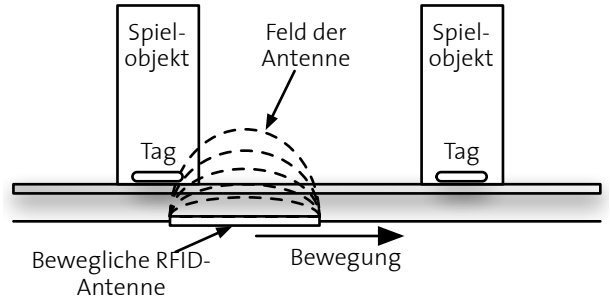


Abbildung 3.2: Prinzip des Spielfelds.

3.2.2. Steuerung der Hardware

Für die Hardware (LEGO Mindstorms, RFID-Lesegerät) sollen objektorientierte Schnittstellen geschaffen werden, die auch in anderen Projekten genutzt werden können. Auch soll es möglich sein, die Schnittstellen durch andere Klassen einfach zu ersetzen, die auf andere Geräte verbinden.

Wo betriebssystemabhängige Bibliotheken zum Ansprechen der Hardware genutzt werden, muss darauf geachtet werden, dass diese Komponenten mindestens unter Linux und OS X lauffähig sind.

3.2.3. Verarbeiten der Daten

Nötige Datenverarbeitungen sollen seriell erfolgen und die einzelnen Module problemlos austauschbar sein. Folgende beiden Module werden bestimmt benötigt.

1. *Mitteln der Messwerte*: Beim Bewegen der Antenne über das Spielfeld werden die Tags mehrmals gelesen. Erst der Mittelwert aller Messwerte ergibt die wahrscheinliche Position des Tags auf dem Spielfeld.
2. *Koordinatentransformation*: Die von der Hardware gelieferten Koordinaten sind nicht in einem für das Spiel geeigneten Format vorhanden und müssen daher erst transformiert werden.

3.2.4. Position und Ausrichtung von Objekten erkennen

In [8] wurde bereits ein Algorithmus für dieses Problem vorgestellt. Dieser berechnet die Abweichung für alle möglichen Winkel und wählt den Winkel mit der kleinsten Abweichung. Es soll eine Methode gesucht werden, die ohne Ausprobieren funktioniert.

Die angestrebten Lesefehler von weniger als $1cm$ würden für dieses Spiel genügend präzise Werte liefern. Zumindest für diesen Prototypen. Bei Turnierspielen wird auch häufiger über wenige Millimeter diskutiert.

Die grösseren Objekte (Fahrzeuge) müssen mit mehr als einem Tag markiert werden, um den Winkel erkennen zu können. Die Fehler der einzelnen Tags werden sich gegenseitig reduzieren, was eine noch bessere Positionsbestimmung zulässt.

3.2.5. Laden von Spielobjekten aus den Konfigurationsdateien

Die Einheiten und Modelle sollen in separaten XML-Dateien abgelegt werden. Dabei ist zwischen der *Vorlage* für ein Modell und dem eigentlichen Modell zu unterscheiden. Möglichst viele Informationen sollten in der Vorlage gespeichert werden und nur die ID des RFID-Tags beim eigentlichen Modell.

Neben den einfachen Modellen müssen auch Fahrzeuge, Waffen und Informationen über das Gelände aus XML-Dateien gelesen werden. Dies soll jeweils über eigene Methoden realisiert werden.

Ein Neuladen der Konfiguration während des Spiels macht keinen Sinn, da die Regeln nicht mitten im Spiel geändert werden dürfen. XML-Schemas werden auch nicht erstellt.

3.2.6. Grafische Benutzerschnittstelle (GUI)

Für oft benötigte Objekte soll ein eigenes Grafikelement (`JComponent`) erzeugt werden. Die einzelnen Spielobjekte werden auf ein virtuelles Spielfeld gezeichnet, das möglichst exakt dem physikalischen Spielfeld entspricht. Der Schwerpunkt liegt auf Funktionalität, nicht auf grafischen Effekten.

Es soll darauf geachtet werden, dass es möglich wäre, ein Foto des echten Spielfelds als Hintergrund des virtuellen Felds zu nutzen.

3.2.7. Spiellogik

Wie bereits erwähnt geht es nicht darum, das Spiel vollständig in Software abzubilden. Das Spiel ist zu vielseitig, um in dieser Arbeit komplett implementiert zu werden, was das rund 290 Seiten starke Regelbuch [6] beweist.

Die Konzepte sollen gezeigt und wenige verschiedene Modelle implementiert werden. Auch Fahrzeuge und Gelände müssen beachtet werden. Gebäude können als eine spezielle Art von Gelände betrachtet werden und bedürfen daher keiner separaten Implementierung.

Die Spiellogik soll in eine Hauptlogik und mehrere Module für die einzelnen Spielphasen unterteilt werden. Dies macht Sinn, da die Regeln in den einzelnen Phasen des Spiels sehr unterschiedlich aussehen.

3.3. Ablaufdiagramme

Für die wichtigsten Spielphasen wurden Sequenzdiagramme erstellt. Die Informationen für diese Diagramme wurden direkt dem Regelwerk von WH40k entnommen und werden hier jeweils nur kurz erläutert.

Die Diagramme enthalten mehr Details und Funktionen als für diesen Prototypen geplant. Der Grund liegt in der geplanten Erweiterbarkeit. Nur wenn man sich bereits bei der Planung mit den zukünftigen Funktionen beschäftigt, lassen sich diese später mit vertretbarem Aufwand integrieren.

Die nachfolgenden Ablaufdiagramme zeigen den Ablauf der Ereignisse im Spiel. Rauten stellen einen Entscheid dar. Häufig werden diese über Würfel entschieden. Dabei bezeichnet zum Beispiel *Roll-to-Hit* eine in den Regeln beschriebene Aktion. Vierecke mit einem zusätzlichen Balken links und rechts bezeichnen Aktionen, die durch ein anderes Ablaufdiagramm beschrieben werden.

Die meisten der folgenden Diagramme sind in der Reihenfolge, wie sie gebraucht werden. Einige einfachere Diagramme wurden auf der gleichen Seite zusammengefasst, um nicht unnötig viele und grösstenteils leere Seiten zu erhalten.

Spielübersicht und Vorbereitung

Abbildung 3.3 zeigt den rundenbasierten Ablauf des Spiels. Trotzdem muss nach jedem Zug überprüft werden, ob jemand gewonnen hat. Es macht keinen Sinn, den zweiten Spieler spielen zu lassen, wenn er keine Figuren mehr hat.

Auf der rechten Seite wird die Vorbereitung des Spieles gezeigt. Die mit dem Kasten zusammengefassten Felder können im Szenario vorgegeben sein. Für diese Arbeit werden diese Eigenschaften über die Konfiguration festgelegt.

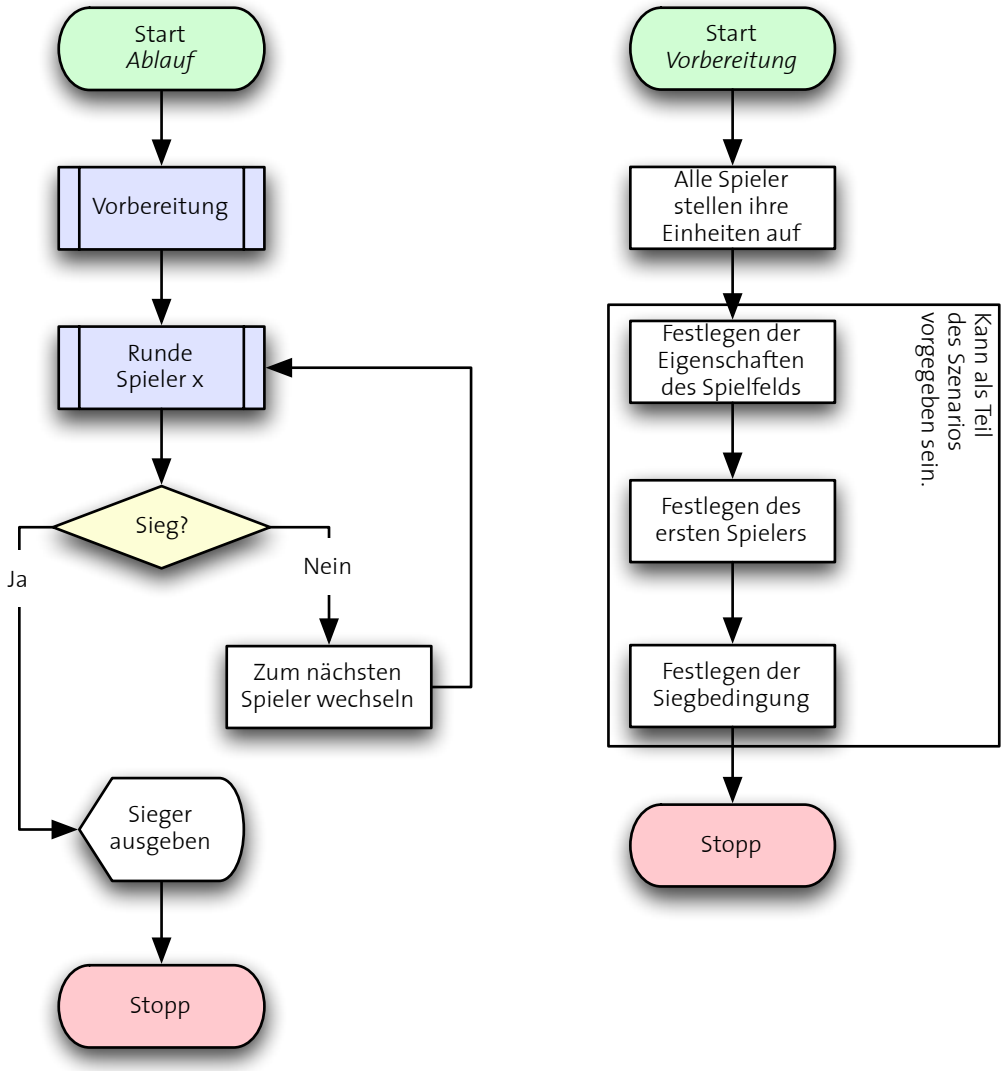


Abbildung 3.3: Spielübersicht (links) und Vorbereitung (rechts).

Rundenübersicht und Feststellen der Richtung und Distanz für Bewegungen

Abbildung 3.4 zeigt die verschiedenen Phasen, die ein Spieler durchgeht, wenn er an der Reihe ist. Die Felder mit zwei Strichen auf der Seite bezeichnen ein weiteres Diagramm, das an dieser Stelle einsetzt.

Auf der rechten Seite wird der Ablauf für das Feststellen der Richtung und Bewegung eines Modells gezeigt. Je nach Art des Modells kann eine Abweichung verlangt sein. Da die maximale Distanz auf schwierigem Terrain kürzer sein kann, muss das hier integriert werden. Die eigentliche Bewegungsphase folgt auf der nächsten Seite.

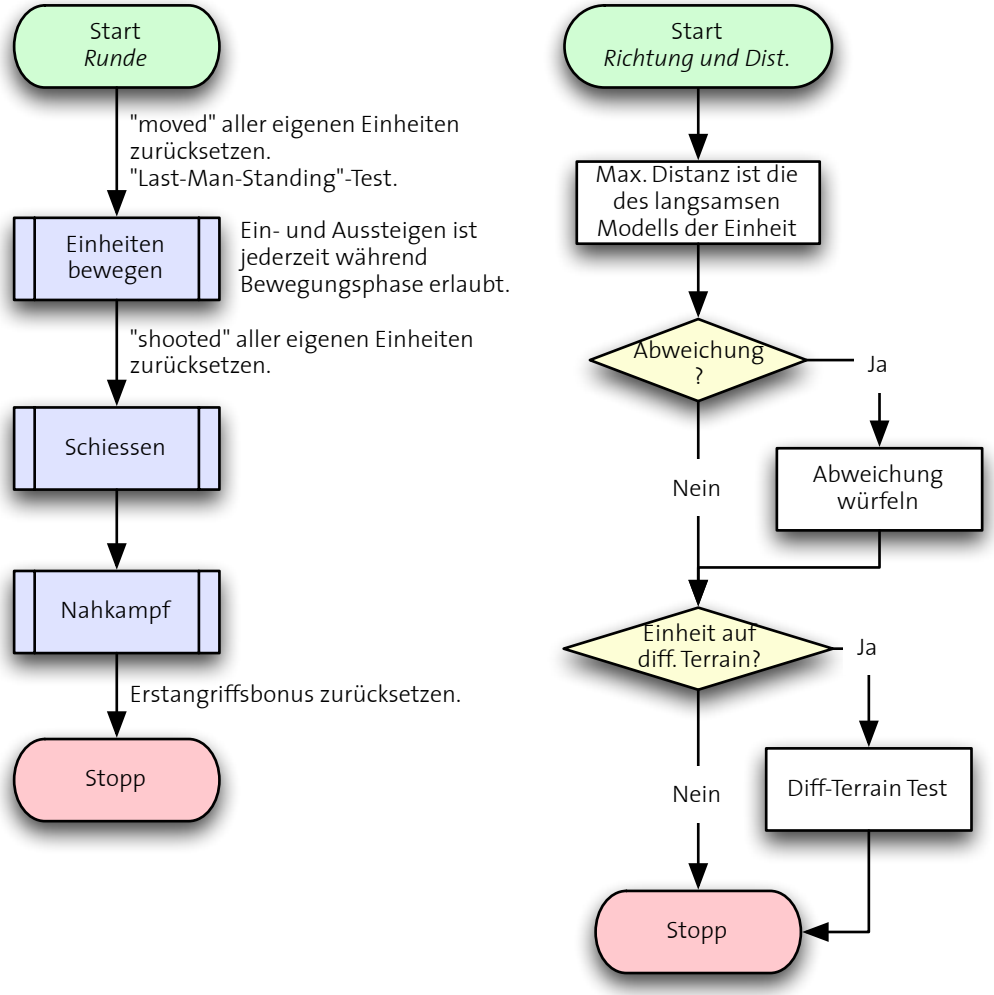


Abbildung 3.4: Rundenübersicht (links), Feststellen der Richtung und Distanz (links).

Bewegungsphase

Abbildung 3.5 zeigt den Ablauf innerhalb der Bewegungsphase. Dazu wird der Ablauf zum Feststellen der Richtung und der Distanz von der letzten Seite genutzt.

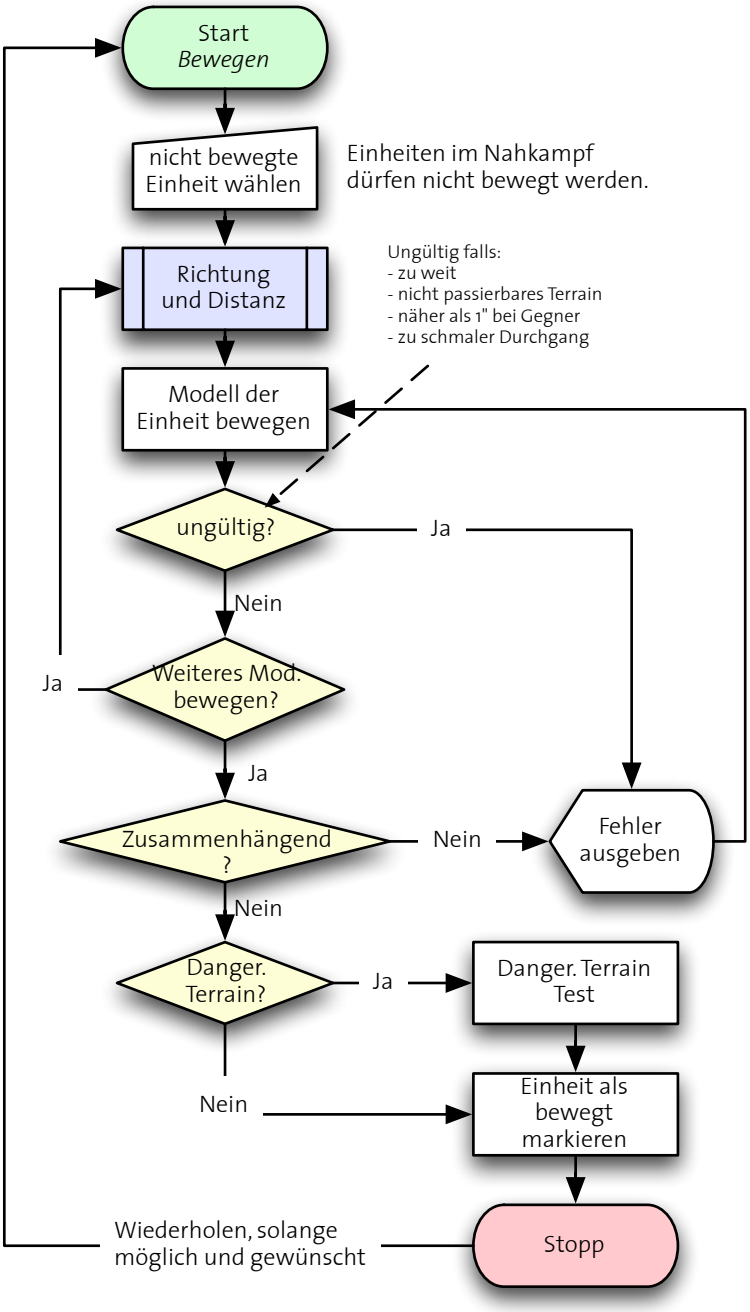


Abbildung 3.5: Bewegungsphase.

Ein- und Aussteigen aus Fahrzeugen

Modelle können in einigen Fahrzeugen transportiert werden. Das Ein- und Aussteigen ist nur in bestimmten Situationen erlaubt. Fahrzeuge und Einheiten dürfen sich nicht in jedem Fall danach noch bewegen.

Das Ein- und Aussteigen wurde für diesen Prototypen nicht implementiert.

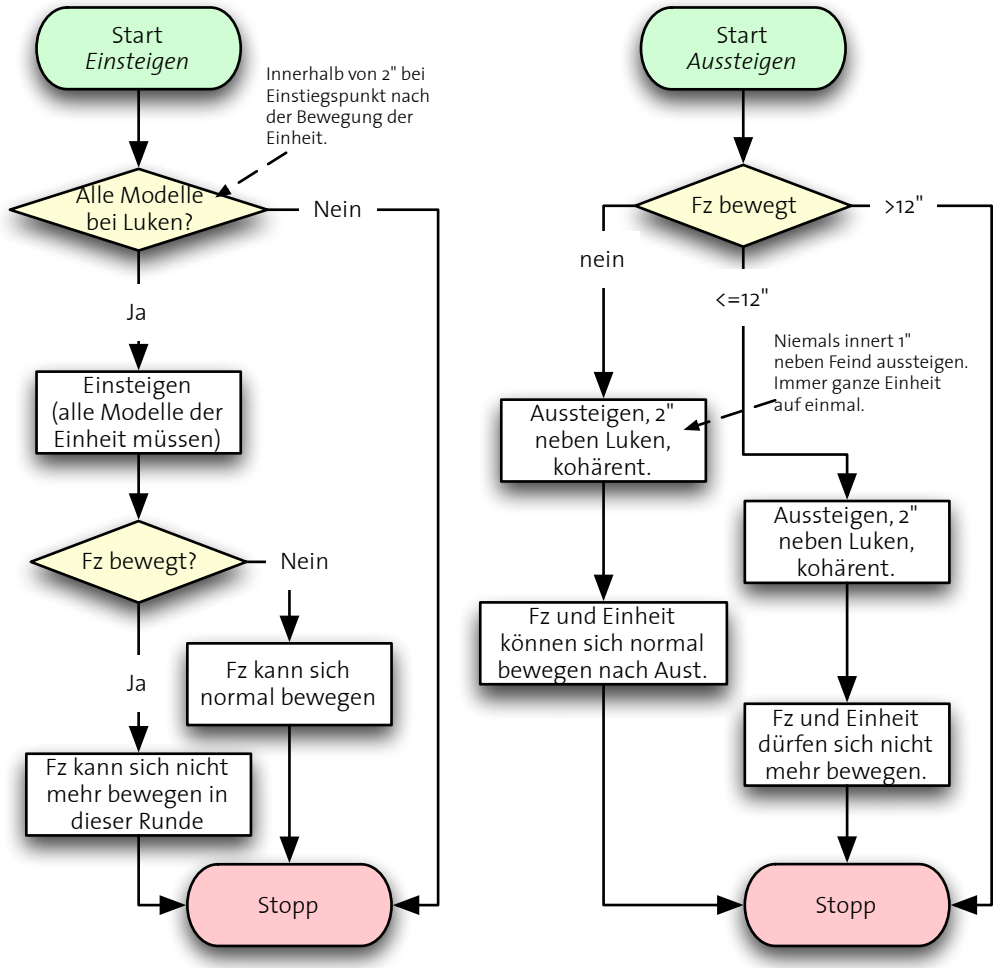


Abbildung 3.6: Einsteigen in Fz (links), Aussteigen (rechts).

Schiessphase

Zuerst wird vom Spieler immer ein Schütze und ein Ziel gewählt. Sollte dieses ausserhalb der Reichweite des Schützen sein, darf dieser in dieser Runde nicht mehr schiessen.

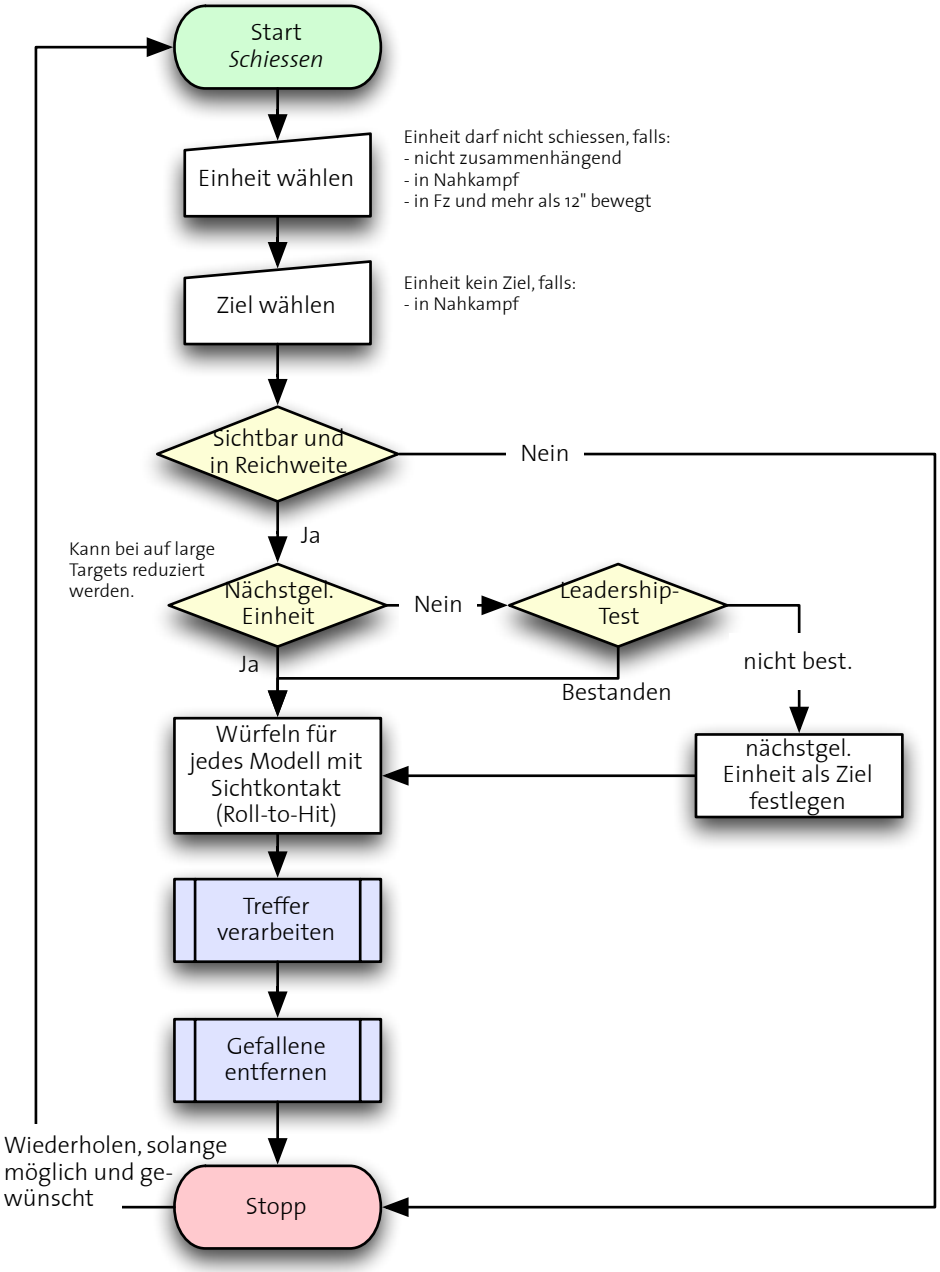


Abbildung 3.7: Schiessphase.

Verarbeiten von Treffern und Entfernen von Gefallenen

In der Schiessphase werden die Anzahl Treffer festgestellt. Dies bedeutet noch keine Wunde für das Opfer. Erst wenn durch weiteres Würfeln eine Wunde zugefügt wird, die nicht durch einen Schutz abgewehrt werden kann, gilt es als zugefügte Wunde.

Stehen die Anzahl Wunden für einen Durchgang fest, werden diese vom Besitzer der Opfer auf seine Modelle aufgeteilt. Sind sehr viele Modelle einer Einheit gefallen, so muss diese Einheit einen Moral-Test bestehen, um nicht zurückzufallen.

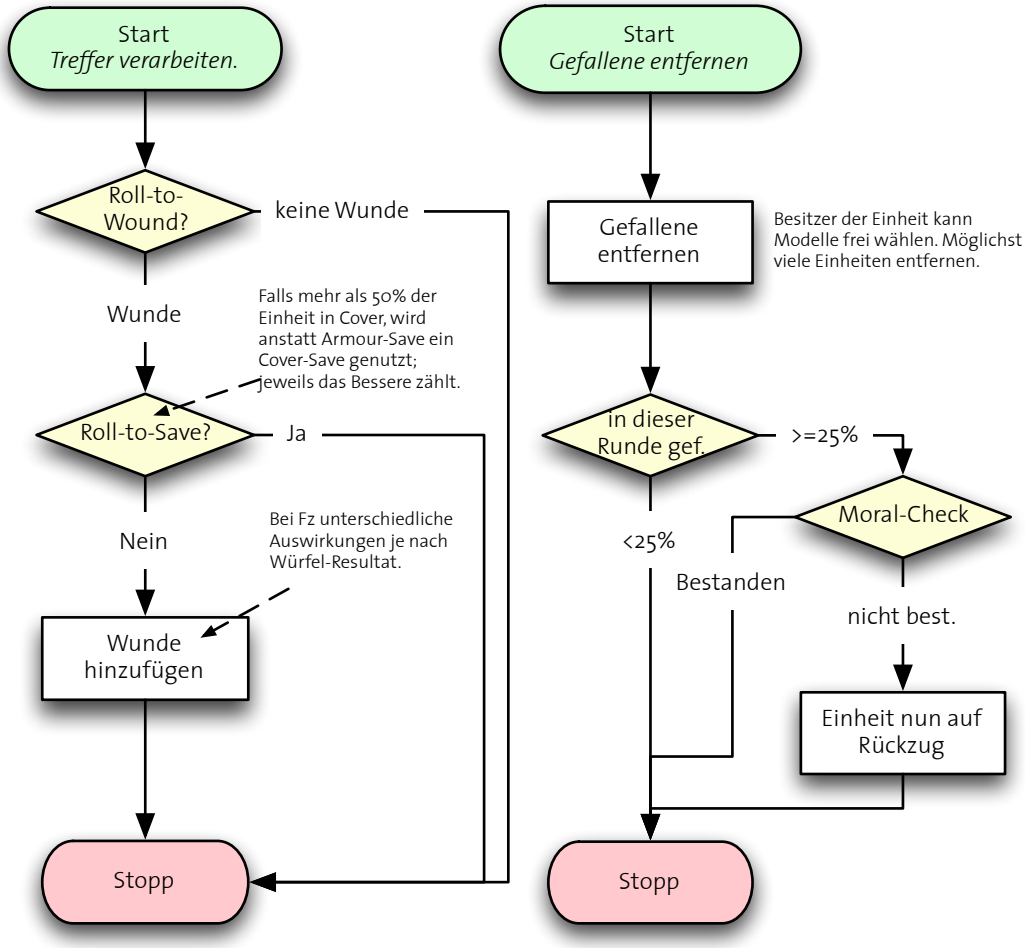


Abbildung 3.8: Verarbeiten von Treffern und Entfernen von Gefallenen.

Nahkampfphase (Übersicht)

Nach dem Schiessen folgt die Phase des Nahkampfs. Hier gelten andere Regeln als während der Schiessphase. So müssen sich die Modelle sehr nahe sein, um in den Nahkampf zu gehen. Auch hier gibt es den Moral-Test für den Verlierer, er hat jedoch andere Auswirkungen.

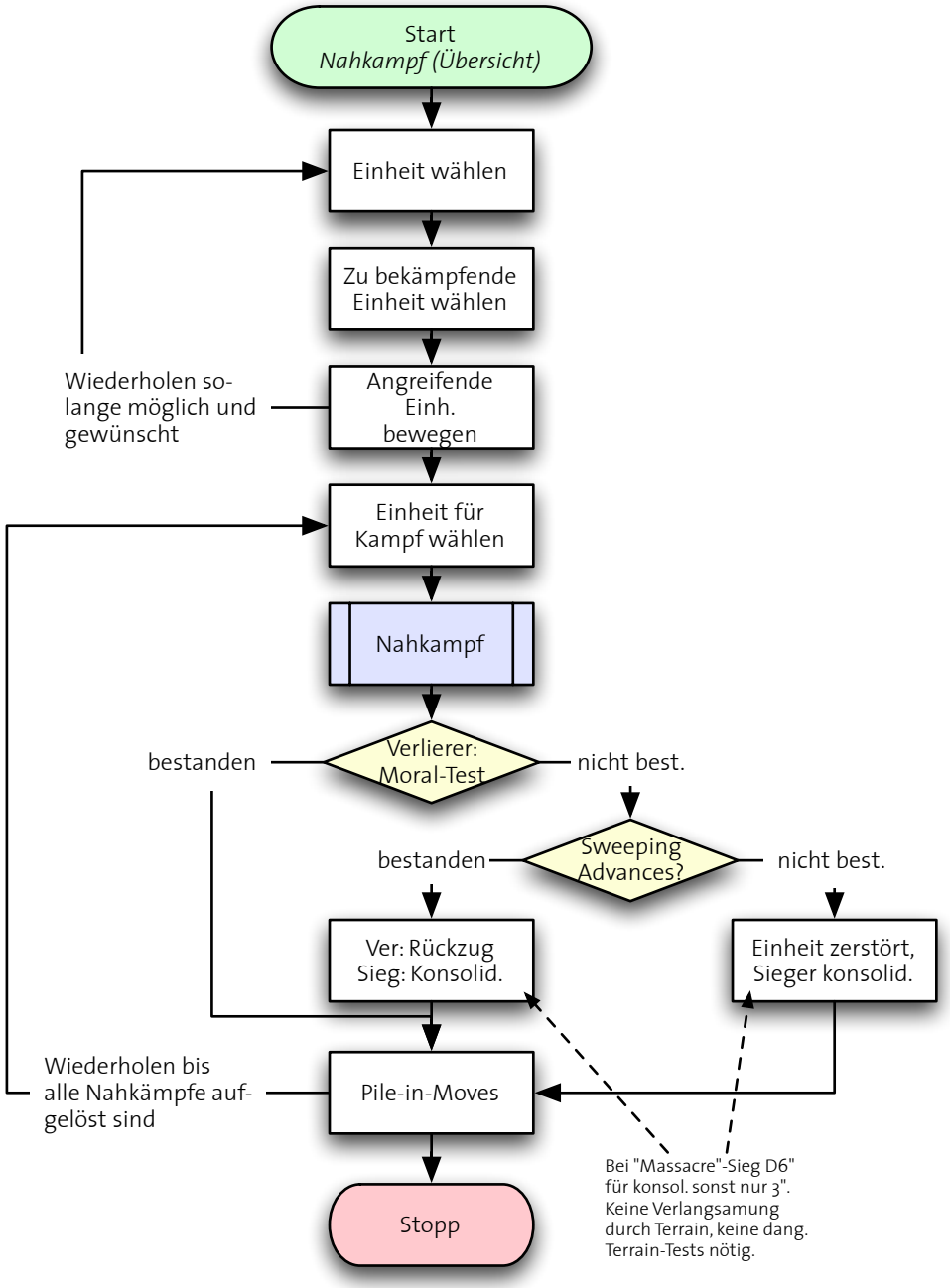


Abbildung 3.9: Übersicht über die Nahkampfphase.

Kampf in der Nahkampfphase

Die eigentliche Kampfhandlung im Nahkampf findet abwechselnd zwischen den beiden betroffenen Spielern statt. Der Initiative-Wert bestimmt dabei die Reihenfolge, in welcher die einzelnen Modelle der Einheit schiessen dürfen.

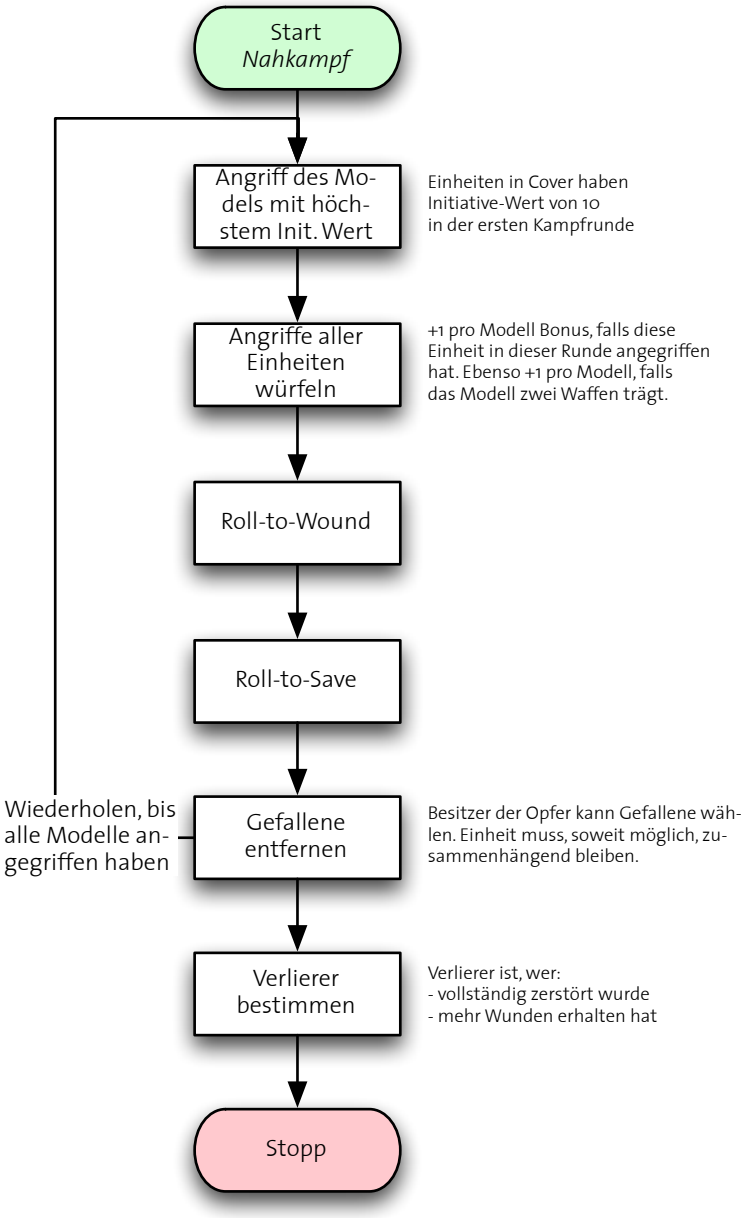


Abbildung 3.10: Kampf in der Nahkampfphase.

3.4. Projektablauf

Aufgrund der in Kapitel 3.2 aufgezählten Module und deren Abhängigkeit ergab sich folgender Ablauf für die Implementierung:

1. Einarbeiten in das Regelwerk
2. Konstruktion des Spielfelds
3. Messen der Genauigkeit und optimieren des Spielfelds
4. Position und Ausrichtung von Objekten mit mehreren Tags bestimmen
5. Benutzerinteraktion festlegen
6. Grundgerüst für grafische Benutzerschnittstelle implementieren
7. Implementieren der Spiellogik und Abschliessen der grafischen Oberfläche
8. Aufräumen und vollständiges Dokumentieren des Quelltexts
9. Schreiben der Dokumentation

Im nächsten Kapitel wird die Umsetzung der einzelnen Module in dieser Reihenfolge beschrieben.

3.5. Schnittstellen für zukünftige Projekte

Auf mögliche Erweiterungen und weiterführende Projekte wird erst im Kapitel 5.2 (Seite 70) eingegangen. Hier werden nur die Vorbereitungen für zukünftige Projekte kurz erwähnt.

Die einzelnen Module sollen so gut als möglich voneinander getrennt werden. Dazu wird die Instanzierung nicht von Hand durchgeführt, sondern ein *Dependency Injection*-Container genutzt. Dieser Begriff bezeichnet eine alternative Methode zum Initialisieren der verschiedenen Module. Anstatt wie traditionell üblich die Klassen selbst alle Abhängigkeiten auflösen zu lassen, wird dies einem Bean-Container übergeben. Dieser bereitet die einzelnen Module (Beans) vor, konfiguriert sie und bindet sie in der richtigen Reihenfolge aneinander. Der Aufwand zum Austauschen eines Moduls reduziert sich auf das Ändern einer einzigen Konfigurationsdatei.

Weiter werden die Spielobjekte sehr stark gekapselt. Der objektorientierte Ansatz ermöglicht es, die Objekte zu erweitern, ohne das Risiko einzugehen das Gesamtsystem negativ zu beeinflussen. Das Aufteilen der Spiellogik in einzelne Module für die verschiedenen Spielphasen macht die Implementierung übersichtlicher, was auch der Erweiterbarkeit dient.

Selbstverständlich wird der Quelltext vollständig dokumentiert, und für die komplexeren Methoden werden Unit-Tests geschrieben.

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Edsger Wybe Dijkstra

4 Umsetzung

Für die Konstruktion des Spielfelds eignet sich LEGO, da sich die Steuereinheit aus dem Robotics-Paket gut von einem PC kontrollieren lässt. Als RFID-Lesegerät wurde ein Exemplar der Firma Feig Electronics (Typ `ID ISC.MR101`) genutzt. Die Wahl der Programmiersprache fiel auf Java. Einerseits existieren viele Module zur Kommunikation mit Hardware, andererseits schränkt man sich nicht auf ein einzelnes Betriebssystem für die Anwendung ein. Für das Ansprechen der Hardware lassen sich Bibliotheken in C oder C++ über JNI (Java Native Interface) einbinden.

Die einzelnen Module aus dem erarbeiteten Konzept wurden übernommen und Stück für Stück implementiert. Zusätzlich musste ein Simulator für das RFID-Spielfeld geschrieben werden, da das echte Spielfeld zu träge ist, um schnelle Testläufe neuer Funktionen zu ermöglichen. Dies führte zu folgenden Teilaufgaben:

- Ansprechen der Hardware
  RFID-Lesegerät und LEGO NXT-Modul
- Konstruieren und Testen des Spielfelds und des Simulators
- Schreiben der Spielplattform
  Verarbeiten der gelesenen Daten, Erkennen von Position und Ausrichtung von Objekten, grafische Darstellung der Daten und Schreiben der Spiellogik

Abbildung 4.1 zeigt den Datenfluss und die wichtigsten Module der Anwendung. Die Daten durchlaufen dabei einige Stationen, auf die auf den nächsten Seiten eingegangen wird. Ganz grob lässt sich der Datenfluss wie folgt beschreiben: Die von der Hardware oder dem Simulator erhaltenen Daten werden erst in ein einheitliches Koordinatensystem transformiert und danach einem Spielobjekt zugeordnet. Das Spielobjekt leitet aus einer oder mehreren erhaltenen Koordinatenangaben seine echte Position ab und meldet jede Änderung der Spiellogik.

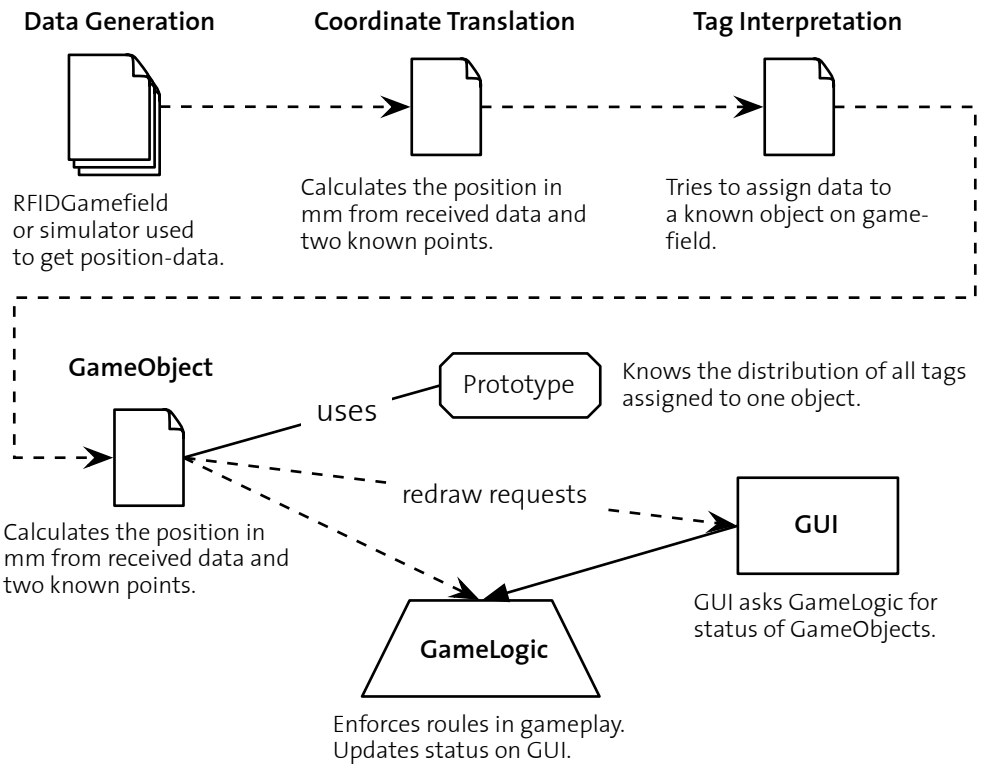


Abbildung 4.1: Grobplan der Architektur.

Zusätzlich wird das GUI angewiesen, dieses Spielobjekt neu zu zeichnen. Zum Zeichnen des Objekts muss das GUI zusätzliche Informationen aus der Spiellogik beziehen.

Die auf Abbildung 4.2 gezeichneten Klassen sind die wichtigsten Komponenten bei der Verarbeitung der Daten, bevor diese dem Spielobjekt übergeben werden. Der `Averager` mittelt die vielen Positionen, die das Spielfeld für jeden einzelnen Tag zurückgibt, bevor der `CoordinateTranslator` die Koordinaten in das spieleigene Koordinatensystem transformiert. Der `TagInterpreter` hat zwei Aufgaben: Erstens muss er die erhaltenen Tags eindeutig einem Spielobjekt zuweisen können, andererseits muss er das Laden weiterer Spielobjekte in Auftrag geben, falls ein Tag gesehen wird, der zu keinem aktuellen Spielobjekt gehört. Die `UnitFactory` und die `WeaponFactory` sind für das Erstellen und Konfigurieren neuer Spielobjekte anhand einiger XML-Dateien zuständig.

Wie die Daten in den Spielobjekten weiterverarbeitet werden, hängt sehr vom Objekt ab. Einfache Modelle besitzen nur einen RFID-Tag, der das Zentrum der Einheit markiert. Grössere Objekte besitzen mehrere Tags. Wie aus diesen Tags die korrekte Position berechnet werden kann, wird im Kapitel 4.4 beschrieben. Die weitere Datenverarbeitung geschieht innerhalb der Spiellogik und wird im entsprechenden Kapitel 4.8 beschrieben.

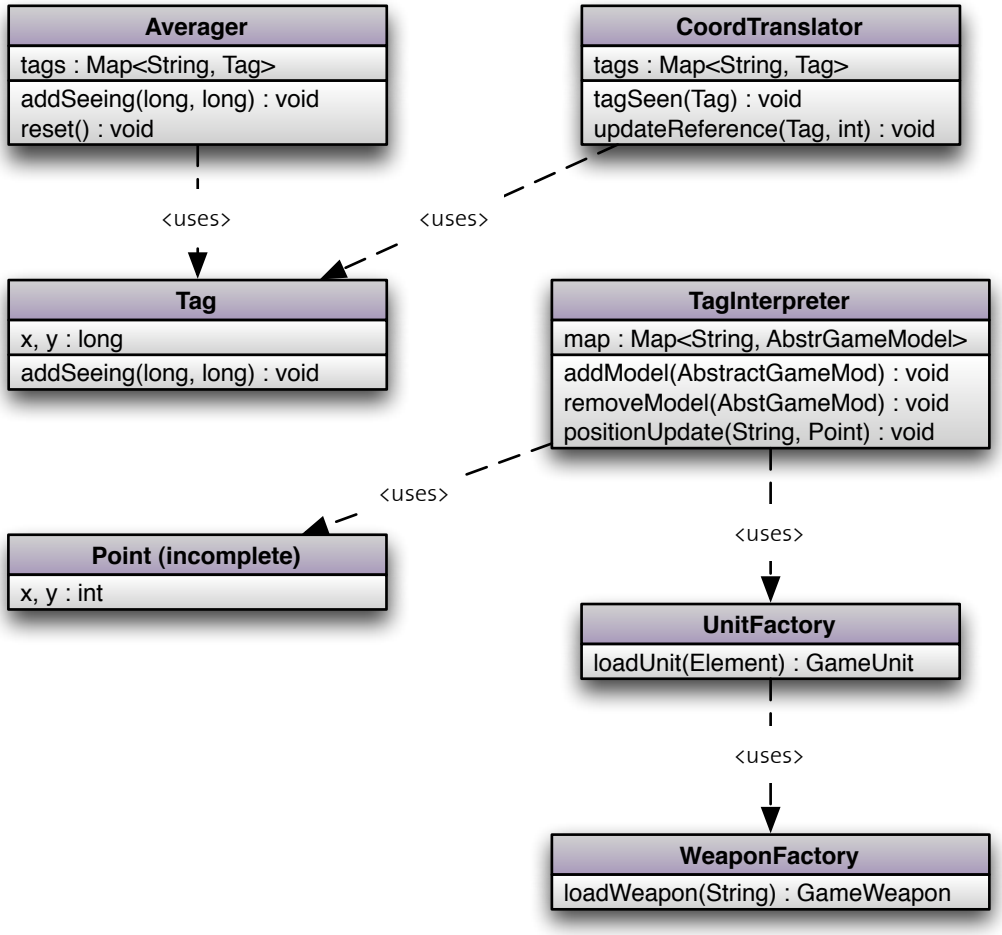


Abbildung 4.2: Klassendiagramm der beim Einlesen von Daten beteiligten Objekte.

4.1. Hardware

Als eine erste Schwierigkeit hat sich die Kommunikation mit der Hardware herausgestellt. Für das Feig RFID-Lesegerät war im Internet keinerlei technische Dokumentation auffindbar. Da eine für Windows geschriebene C-Bibliothek aus einer anderen Arbeit [8] genügend Daten enthielt, wurde jedoch kein Kontakt mit der Firma Feig Electronic gesucht.

Für das LEGO NXT-Modul ist viel technische Dokumentation vorhanden, da dieses Modul sehr häufig für selbstgebaute Roboter genutzt wird. Nicht selten wird die Steuerung dieser Roboter einem PC überlassen. Nur lässt sich das LEGO NXT-Modul nicht wie das Feig-Lesegerät über RS-232 ansprechen, sondern benötigt Bluetooth oder USB-Kommunikation. Das Ansprechen dieser Hardware aus Java gestaltete sich schwierig.

Abbildung 4.3 zeigt die verschiedenen Module, die zum Steuern des Spielfelds und der Datenverarbeitung genutzt werden sowie die Schnittstellen zu externer Hardware.

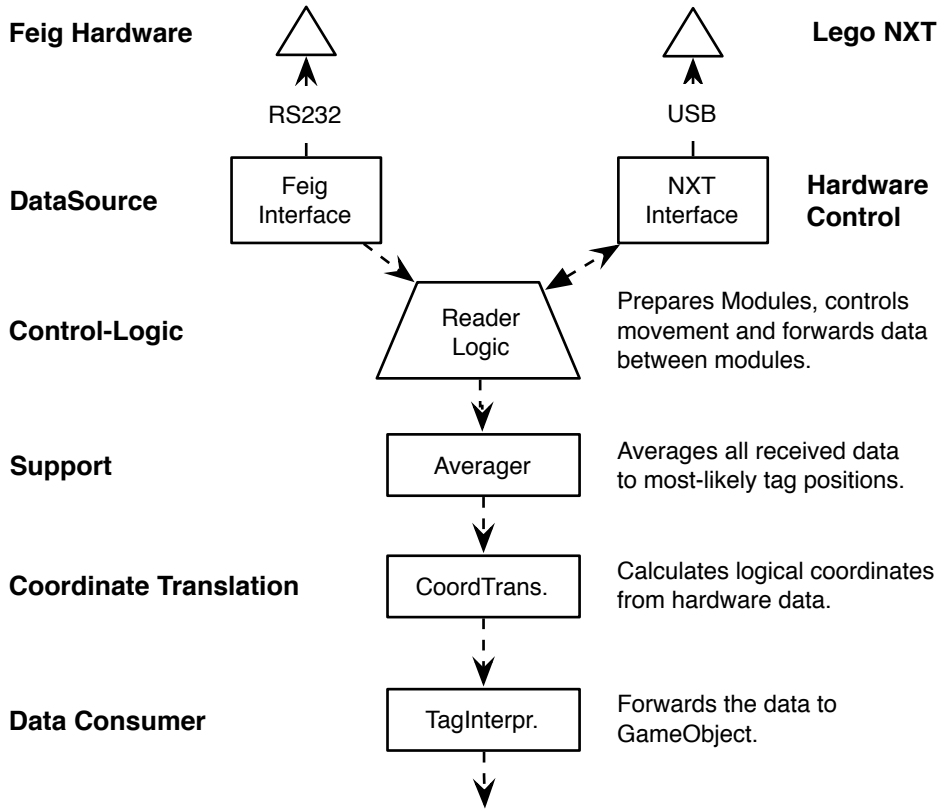
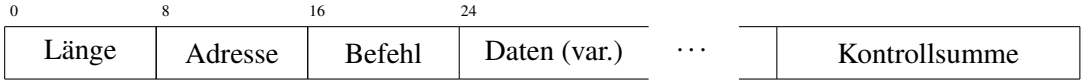


Abbildung 4.3: Architektur RFID-Spielfeld und Datenverarbeitung.

4.1.1. RFID Lesegerät

Das Feig RFID-Lesegerät vom Typ `ID ISC.MR101` wird über RS-232 an den PC angeschlossen und benutzt ein proprietäres Protokoll. Aus einem C-Programm [8] konnten sämtliche benötigten Informationen extrahiert werden.

Das Lesegerät benötigt einzelne Befehlsblöcke, die immer identisch aufgebaut sind. Erst kommt ein Byte, das die Länge der Nachricht enthält, danach eine Kommunikationsadresse, die fest auf `0xff` gesetzt wird, ein Befehlsbyte, beliebig viele Datenbytes und eine zwei Byte lange Kontrollsumme.



Die Antwort ist identisch aufgebaut. Erst die Länge, danach werden Kommunikationsadresse und Befehlsbyte wiederholt. Es folgen die Datenbytes und zwei Bytes mit der Prüfsumme. Für diese Arbeit beschränkte sich die Kommunikation mit diesem Gerät auf einen Befehl zum Zurücksetzen (`0x69`) und einen zweiten Befehl zum Lesen sämtlicher Tags in Reichweite (`0xb0`).

Obwohl RS-232 eher unproblematisch ist, erschwerten Probleme mit dem verwendeten USB-Seriell-Adapter die Entwicklung. Nach jedem Lesevorgang musste dieser Adapter vom PC getrennt und neu verbunden werden. Nach langer Suche stellte sich heraus, dass nicht die selbst geschriebene C-Klasse die Probleme verursachte, sondern der Treiber zum USB-Seriell-Adapter. Ein alternativer Treiber [7] funktionierte einwandfrei.

Bei diesem Projekt ist es notwendig, so schnell als möglich zu lesen. Leider scheint die Firmware auf dem Lesegerät mit zu kurz aufeinanderfolgenden Anfragen nicht klarzukommen. Nach wenigen Sekunden reagierte das Gerät auf keine Anfragen mehr und musste von Hand zurückgesetzt werden. Nur durch Ausprobieren konnten an einigen Stellen kurze Pausen eingebaut werden. Dies führt zu den in Tabelle 4.1 und Abbildung 4.4 aufgezeigten Leistungen. Die eingebaute Verzögerung beträgt nur $10ms$ pro Lesevorgang und beeinflusst daher die Resultate kaum. Mehr als 8 Tags konnten nicht getestet werden, da das Lesegerät bei zu vielen Tags in Reichweite nicht mehr jeden zuverlässig erkennt.

Tabelle 4.1.: Leistung des Feig RFID-Lesegerätes beim Einlesen von Tags

Anzahl Tags in Reichweite	Benötigte Zeit $[ms]$	
	Bereich	Schnitt
kein Tag	136-161	154
1 Tag	173-200	189
2 Tags	177-201	195
4 Tags	188-211	205
6 Tags	229-251	244
8 Tags	439-461	455

Man sieht deutlich, dass die benötigte Zeit pro Lesevorgang bei mehr Tags in Reichweite schnell ansteigt. Die Zahlen variieren bei diesem statischen Testzustand wenig. Es hat sich jedoch gezeigt, dass beim Bewegen der Antenne bis doppelt so viel Zeit pro lesung benötigt wird. Diese Zahlen beeinflussen direkt die maximale Geschwindigkeit für die Bewegung der Antenne unter dem Spielfeld.

$$v_{max} = 2\frac{prec}{time} \tag{4.1}$$

Die Geschwindigkeit ergibt sich aus der gewünschten Präzision ($prec$), der Zeit pro Lesevorgang ($time$) und dem Faktor 2, da die Tags bei mehreren Lesevorgängen gesehen werden. Wird ein Tag zum Beispiel zwei Mal gelesen, liegt der Aufenthaltsort zwischen den beiden Lesepunkten. Das verdoppelt die Auflösung.

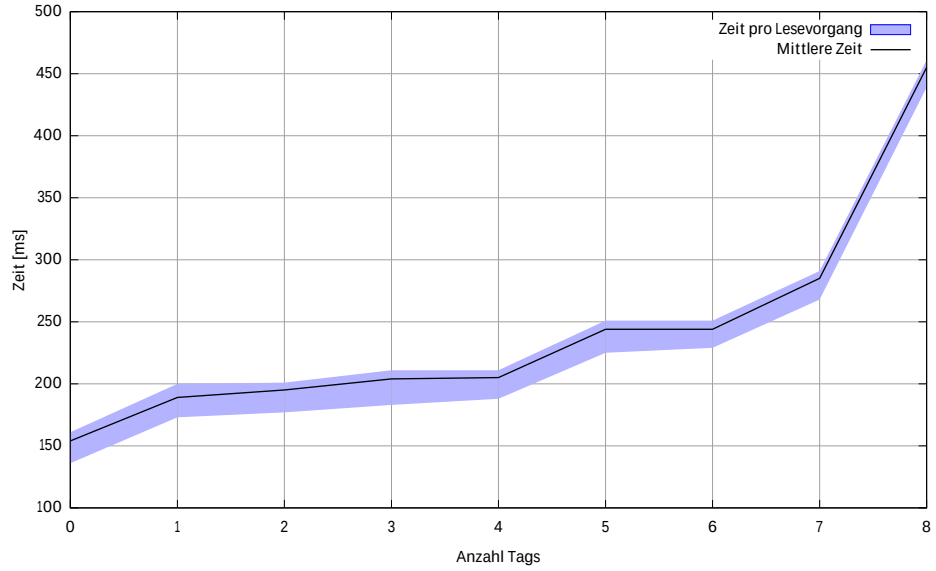


Abbildung 4.4: Benötigte Zeit zum Lesen von Tags.

Die obigen Messungen wurden direkt aus einem in C geschriebenen Programm erstellt. Für die Integration mit Java wurden über JNI nur folgende Funktionen an Java weitergegeben:

```
JNIEXPORT jboolean JNICALL Java_ch_ethz_lthomas_ma_feig_FeigJNI_init_1hw
        (JNIEnv *env, jobject jobj, jstring jdrv) {

JNIEXPORT jstring JNICALL Java_ch_ethz_lthomas_ma_feig_FeigJNI_scan_1field
        (JNIEnv *env, jobject jobj) {
```

Die erste Methode dient zum Initialisieren der Hardware und benötigt die Adresse der seriellen Schnittstelle (zum Beispiel `/dev/ttyS0`)als Argument. Die zweite Methode liest alle Tags in Reichweite und gibt die IDs als Zeichenfolge, getrennt mit „#”, zurück.

Auch die Java-Klasse `FeigJNI` bietet nicht mehr Methoden an. Wird die Klasse initialisiert, beginnt ein asynchroner Verbindungsaufbau. Die Methode `scan` ist ebenfalls ein Wrapper für die entsprechende C-Funktion. Dabei wird jedoch ein String-Array zurückgegeben anstatt der Liste der Tags. Messungen haben gezeigt, dass durch JNI und die Umwandlung in ein Array nur ca. $15ms$ zusätzlich benötigt werden pro Lesevorgang.

4.1.2. LEGO NXT-Modul

Erste Versuche mit dem LEGO RCX-Modul aus frühen LEGO Mindstorms-Sets schlugen fehl. Die Motoren sind zu unpräzise und die Infrarot-Kommunikation zu unzuverlässig. Das Nachfolgemodell (NXT) löst diese Probleme.

- Die Motoren besitzen eingebaute Winkelmesser und können auf wenige Grad genau bewegt werden.
- Die Kommunikation zwischen NXT und PC erfolgt über Bluetooth oder USB.

Eine Kommunikation über Bluetooth hat den Vorteil, dass kein weiteres Kabel benötigt wird. Die vorhandenen Schnittstellen in Java erleichterten die Arbeit zusätzlich. Bluetooth stellte sich jedoch später als unbrauchbar heraus, da die Laufzeit der Nachrichten zu lang ist. Laufzeiten von $200 - 250ms$ machen eine präzise Steuerung sehr schwierig.

USB ermöglicht Laufzeiten von $3 - 5ms$ und ist dadurch schnell genug um genaue Werte zu erhalten. Die Integration der USB-Schnittstelle in Java erfolgte über eine weitere JNI-Klasse. Der native C-Teil konnte vom LEJOS-Projekt [3] übernommen werden. Die dort enthaltene `pccomm.jar` enthält grundlegende C-Funktionen, um mit dem NXT-Modul zu kommunizieren. Die im selben Projekt veröffentlichte iCommand-Bibliothek, die eine objektorientierte Schnittstelle zum NXT-Modul anbietet, war bei Projektstart noch nicht auf OS X lauffähig. Eine eigene Entwicklung, die exakt die benötigten Funktionen abbildet, wurde daher geschrieben (Abbildung 4.5). Diese Klassen bieten eine einfache objektorientierte Schnittstelle zu den Motoren und Sensoren des NXT-Moduls. Zur Kommunikation mit dem NXT wird keine Kontrollsumme benötigt, die Daten werden jedoch binär übertragen. Die entsprechende Umwandlung, das Versenden der Daten und das Lesen der Antwort wurde in diese Klassen verpackt. Details zum Kommunikationsprotokoll konnten der ausgezeichneten Dokumentation von LEGO [4] entnommen werden.

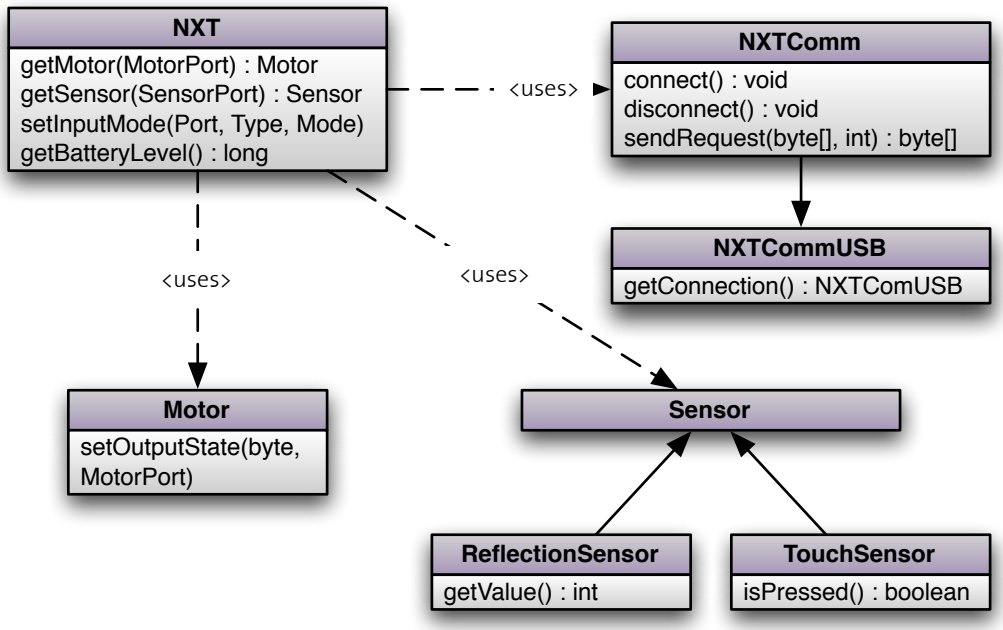


Abbildung 4.5: Klassendiagramm der NXT-Schnittstelle.

Die Steuerung funktioniert grundsätzlich sehr zuverlässig. Einziges Problem sind die Motoren. Teilweise haben sie Mühe, einen gewünschten Wert exakt zu erreichen. Wird die Geschwindigkeit für die Bewegung zu tief eingestellt, bewegt sich der Motor überhaupt nicht, bei einem zu hohen Wert dreht er häufig zu weit. Speziell bei schwereren Modellen scheint die bewegte Masse den Motor über den eingestellten Zielwert hinaus zu ziehen. Dies konnte teil-

weise korrigiert werden durch Abbremsen vor dem Erreichen des Ziels und einer langsamer ausgeführten Korrekturbewegung. Weitere Umbauten am Spielfeld könnten diese Problematik weiter reduzieren.

Häufig schlägt der erste Verbindungsversuch mit dem NXT-Modul fehlt oder es werden keine korrekten Werte zurückgegeben. Der Fehler liegt in der C-Bibliothek von LEJOS unter OS X und ist den Entwicklern bekannt, konnte jedoch noch nicht behoben werden. Das Problem konnte ein wenig entschärft werden durch Verbindungstests direkt nach dem Verbindungsaufbau. Der Ladezustand der Batterien wird so oft abgefragt, bis gültige Werte zurückgegeben werden. In einigen Fällen hilft jedoch nur ein Neustart des NXT-Moduls.

4.2. Spielfeld

Das Spielfeld ist ein sehr wichtiger Bestandteil dieser Arbeit. Einerseits sollte die nutzbare Fläche so gross wie möglich sein, das Einlesen darf jedoch nicht zu lange dauern. Eine einzelne Runde bei WH40k dauert meistens über 15 Minuten. Trotzdem ist es wichtig, dass das GUI schnellstmöglichst die aktuellen Daten anzeigt. Da es sich nur um einen Prototypen handelt wurde die Spielfeldgrösse nach einigen praktischen Tests auf $40x40cm$ beschränkt.

Wegen der RFID-Technologie dürfen auf dem Spielfeld keine metallischen Gegenstände sein. Hochwertigere Modelle bestehen häufig aus Metall und werden daher nicht immer korrekt erkannt. Während unter Plastikfiguren montierte Tags beim Einlesen des Spielfelds ungefähr 30 Mal erkannt werden, passiert das bei solchen unter Metallfiguren nur 10-15 Mal. Welchen Einfluss dies auf die Genauigkeit der Positionserkennung hat, wurde nicht untersucht.

Um dieses und ähnliche Risiken auszuschliessen, wurde speziell darauf geachtet, dass nichts Metallisches in der Nähe der RFID-Antenne sein kann. Es wurden bei den Tests ausschliesslich Plastikfiguren verwendet. Das Spielfeld bestand aus einem auf einem Holztisch montierten Stück Karton und LEGO-Elementen aus Plastik.

4.2.1. Konstruktion

Die Antenne muss möglichst präzise unter dem Spielfeld geführt werden. Dabei darf sich nichts verkanten und die Kabel dürfen nicht blockieren. Ein erster Versuch ist in Abbildung 4.6 dargestellt.

Es handelt sich um zwei Wagen auf der Seite, die jeweils mit einem Motor ausgerüstet sind, um die Bewegung auf der X-Achse sicherstellen. Zwischen diesen beiden Wagen befindet sich eine Schiene, auf der ein Schlitten die RFID-Antenne bewegt. Dieser Schlitten ist mit einem dritten Motor ausgerüstet, der die Bewegung auf der Y-Achse ausführen kann. Darüber liegt das Spielfeld. Das Spielfeld musste ungefähr $60x50cm$ breit gebaut werden, um einen nutzbaren Bereich von $40x40cm$ zu erhalten. Dieser zusätzliche Platz wird von der Antenne benötigt. Erst wenn die Antenne so weit fährt, dass kein Modell mehr erfasst wird, kann sichergestellt werden, dass alle Modelle exakt vermessen wurden.

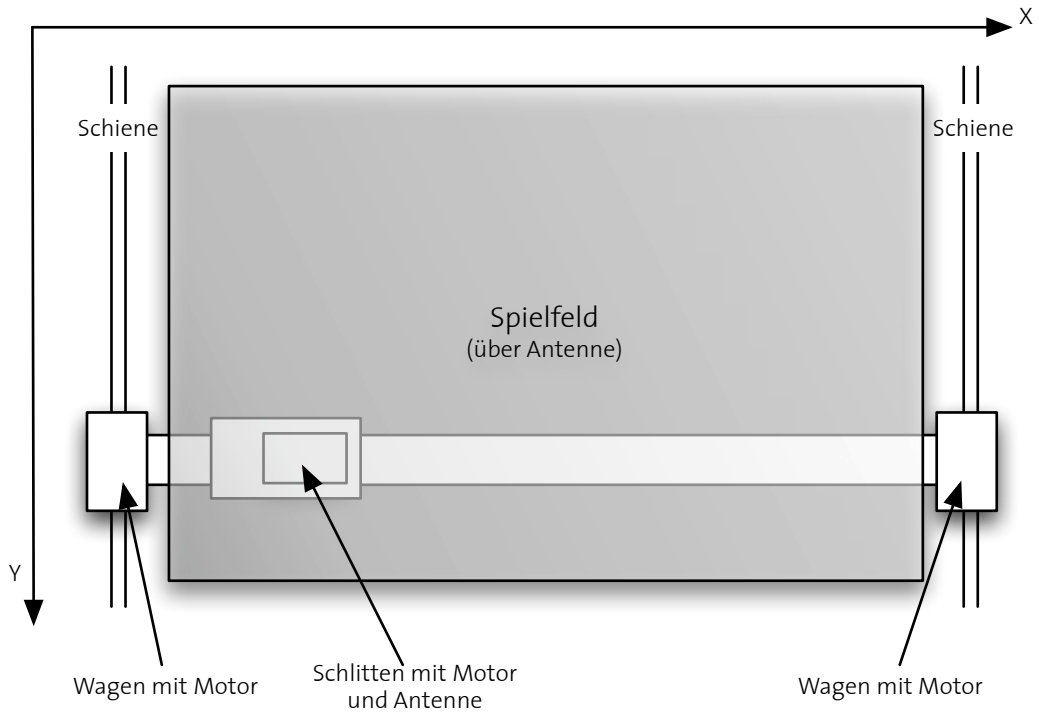


Abbildung 4.6: Die erste Version des RFID-Spielfelds.

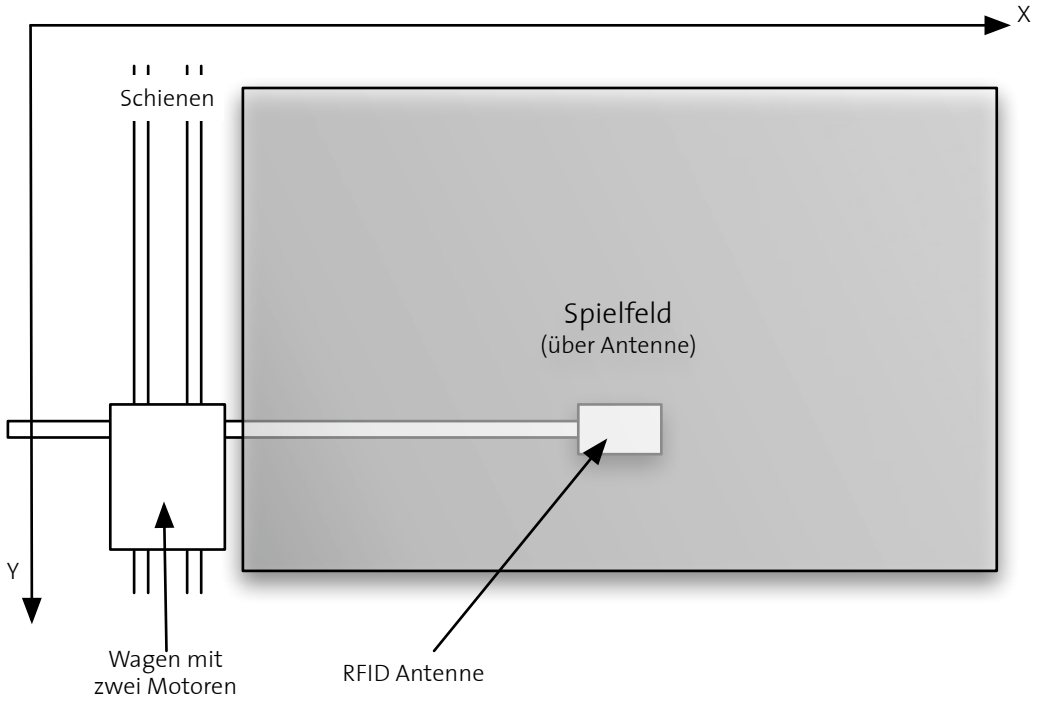


Abbildung 4.7: Überarbeitetes RFID-Spielfeld.

Bei dieser Version hat die Nähe von Motor und RFID-Antenne die Empfangsqualität so stark beeinflusst, dass das System nicht mehr nutzbar war. Es musste eine andere Möglichkeit gefunden werden, bei der der Motor weiter weg von der Antenne montiert ist. Diese Überlegung führte zu dem Spielfeld aus Abbildung 4.7.

Diese überarbeitet Version hat einen einzelnen Schlitten auf der linken Seite des Spielfelds. Auf diesem sind Motoren für die X- und die Y-Achse untergebracht. Bewegungen auf der Y-Achse werden durch Bewegen des ganzen Schlittens ausgeführt. Die X-Achse kann mit einem langen Arm abgefahren werden. An der Spitze dieses Arms befindet sich die RFID-Antenne.

Die Distanz von Motor und Antenne ist jederzeit genügend gross. Durch diesen langen Arm, der unter der Antenne mit einem kleinen Rad gestützt werden musste, sind jedoch Bewegungen auf der Y-Achse nur noch möglich, wenn der Arm ganz nach links gefahren ist. Ist der Arm ganz ausgefahren, bleibt dieser bei der Y-Bewegung auf der aktuellen Höhe stehen und liefert unbrauchbare Werte beim Zurückfahren. Ein weiterer Nachteil dieser Lösung ist der lange Arm, der auf einer Seite des Spielfelds ausfährt. Läuft ein Spieler in den Arm hinein bricht die LEGO-Konstruktion auseinander. Abbildung 4.8 zeigt die fertige Konstruktion und Abbildung 4.9 ein gerendertes 3D-Modell aus einem anderen Blickwinkel.

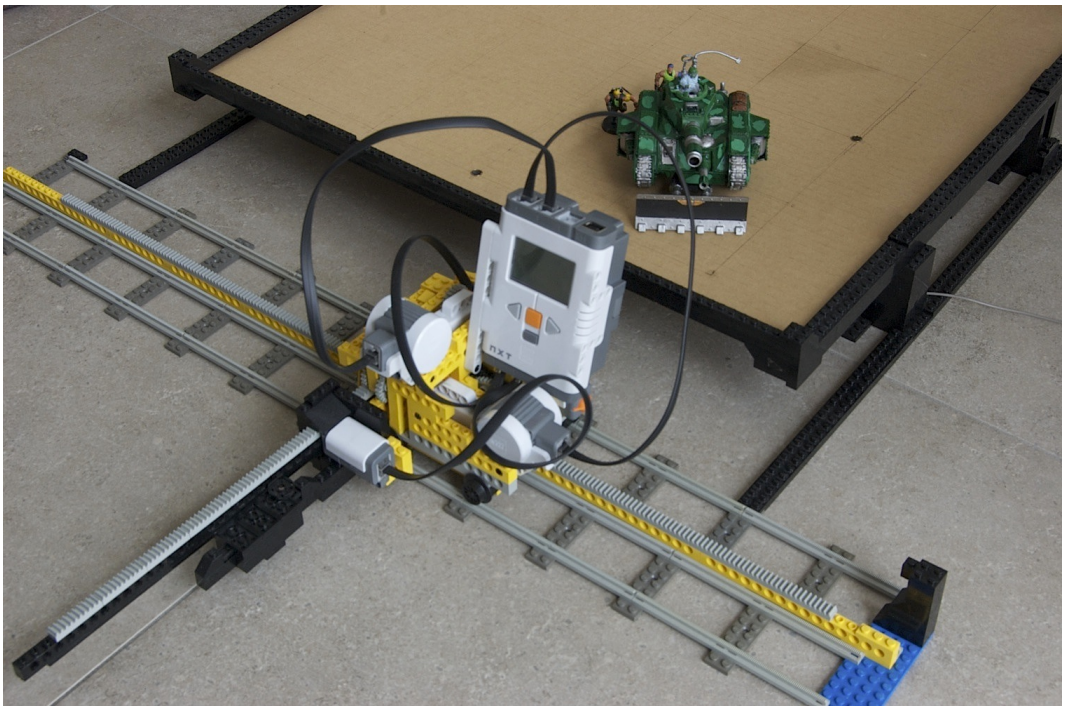


Abbildung 4.8: Foto der endgültigen Version des Spielfelds.

Für grössere Spielfelder könnten mehrere solche Konstruktionen genutzt werden, wodurch sich die Zeit für einen einzelnen Lesedurchgang nicht weiter erhöhen würde. Alternativ wäre auch das Hinzufügen weiterer Antennen auf die selbe LEGO-Konstruktion möglich, um die

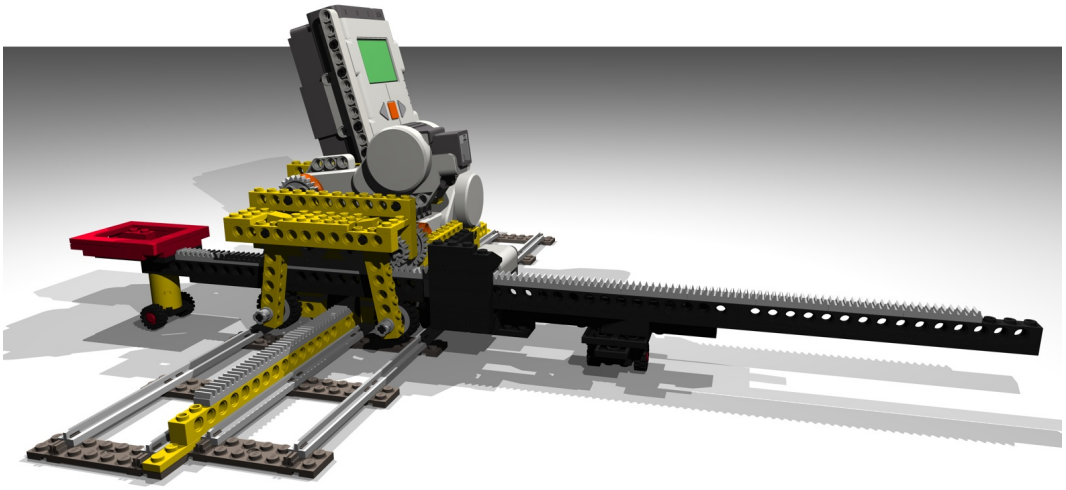


Abbildung 4.9: 3D-Modell der endgültige Version des Roboters.

Zeit zu reduzieren. Dazu darf jedoch kein Multiplexer genutzt werden, da dieser zu viel Zeit benötigt um zwischen den Antennen zu wechseln und dadurch die Leseleistung reduziert.

Die Modelle wurden mit Nano-Size Tags der Firma Tagsys markiert (Abbildung 4.10, rechts). Während der Tests ist kein einziger Tag ausgefallen und daher wurde auch nicht auf Redundanz geachtet. Da die Symmetrie gewahrt werden muss, ist es relativ schwierig, mehrere Tags an einem kleinen Modell anzubringen. Es hat sich gezeigt, dass die Nano-Size Tags bei einem Lesevorgang zwischen 25 und 30 Mal gelesen wurden, die Small-Size Tags 100 bis 120 Mal. Eine ins Spielfeld integrierte, vier auf drei Zentimeter grosse Antenne von Feig (`ID ISC.ANT 40/30`) ermöglichte dem RFID-Lesegerät das Erfassen der Tags in deren Reichweite.

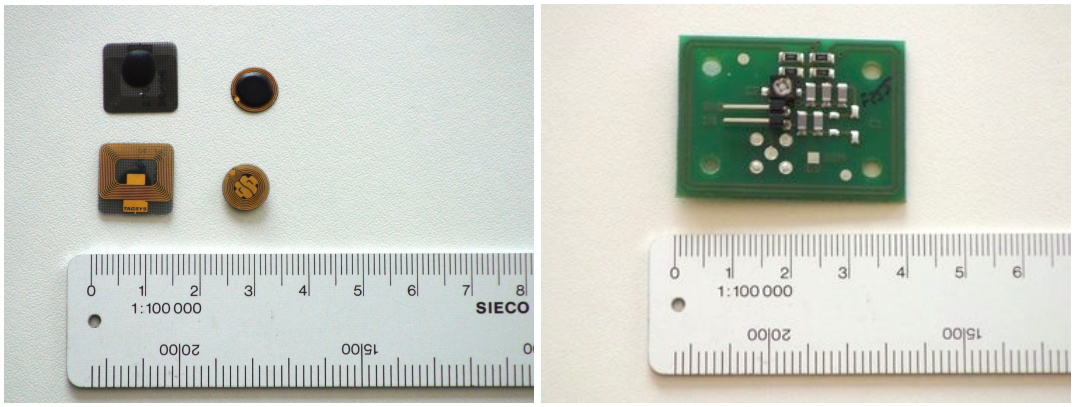


Abbildung 4.10: Zwei Grössen RFID-Tags und die genutzte Antenne.

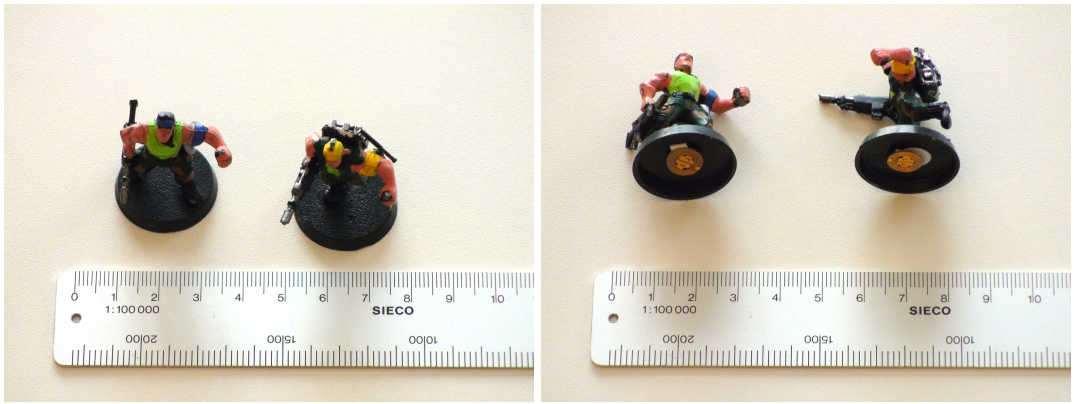


Abbildung 4.11: Mit RFID-Tags versehene Modelle.

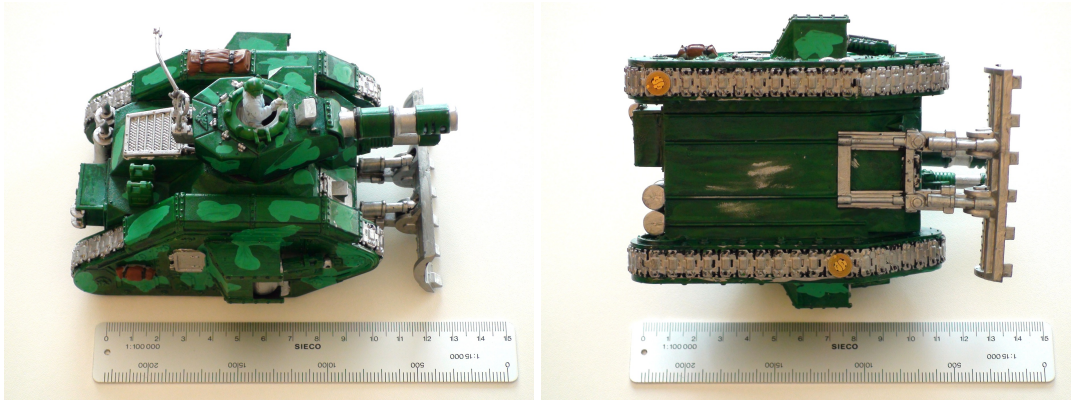


Abbildung 4.12: Mit RFID-Tags versehenes Fahrzeug.

Die Figuren erhielten jeweils einen Tag in der Mitte des Fusses (Abbildung 4.11). Die Fahrzeuge wurden mit zwei Tags ausgerüstet, die soweit als möglich voneinander entfernt angebracht wurden (Abbildung 4.12). Dadurch kann die Ausrichtung der Modelle besser erkannt werden. Auch wo genügend Platz vorhanden war, wurden nur die Nano-Size Tags genutzt. Diese Tags sind symmetrischer aufgebaut als die grösseren, was vermutlich zu besseren Resultaten führt. Dies wurde jedoch nicht getestet.

4.2.2. Steuerung

Bei der Initialisierung des Spielfelds fährt der Arm langsam zurück, bis der Reflektionssensor auf die im Arm integrierten weissen LEGO-Steine trifft. Der Sensor ist nicht sehr präzise und so kann die Antennenposition ca. $5mm$ variieren. Durch die Referenztags auf dem Spielfeld spielt dies jedoch keine Rolle, da diese Verschiebung dadurch ausgeglichen wird. Weitere Informationen zu diesen Referenztags folgen im Kapitel Koordinatentransformation. Da der Arm eingefahren ist, darf sich der ganze Schlitten auf der Y-Achse bewegen. Er bewegt sich so lange zurück, bis der Drucksensor das Ende der Schiene berührt. Das ganze System befindet sich nun ungefähr an der richtigen Startposition und der erste Scan-Vorgang kann beginnen.

Der Arm wird ausgefahren und dabei schnellstmöglich das RFID-Lesegerät abgefragt. Zwischen den einzelnen Lesevorgängen des Readers wird der LEGO-Motor nach dem aktuellen Drehwinkel gefragt. Wird ein Tag gelesen, werden die aktuellen Drehwinkel beider Motoren als X- und Y-Koordinate genutzt. Da jeder Tag mehrere Male gelesen wird, müssen alle Werte des selben Tags gemittelt werden. Hat der Arm die maximale Reichweite erreicht, wird dieser gestoppt und wieder zurückgefahren. Dies ist notwendig, da diese Konstruktion keine Bewegung des Schlittens zulässt, wenn der Arm ausgefahren ist. Nun bewegt sich der ganze Schlitten einige Millimeter nach vorne um danach den Arm an der neuen Position auszufahren. Diese Schritte werden wiederholt, bis der maximale Y-Wert erreicht wird. Der ganze Schlitten fährt danach wieder an die Ursprungsposition und beginnt erneut (Abbildung 4.13, Variante a). Das einmalige Scannen eines $40x40cm$ grossen Spielfelds dauert so knapp über 10 Minuten.

Optimierungen

Das präzise Abtasten des ganzen Spielfelds braucht viel Zeit. Wird die Bewegung des Arms beschleunigt, sinkt die Auflösung auf der X-Achse. Werden die Schritte des Schlittens vergrössert, sinkt die Auflösung auf der Y-Achse. Dies führte zu verschiedenen Optimierungen des Steuer-Algorithmus.

Eine erste Optimierung bestand aus dem Beschleunigen des Ausfahrens des Arms (Abbildung 4.13, Variante b). Da während dieser Bewegung keine RFID-Tags gelesen werden, spielt die Auflösung keine Rolle und die maximale Geschwindigkeit kann genutzt werden. Die Dauer eines Durchgangs konnte so um fast 20% reduziert werden.

Noch schneller lässt sich das Spielfeld abfragen, wenn die bisher ungenutzte Zeit während dem Ausfahren des Arms auch für Lesevorgänge benutzt wird. Scannt man beim Ausfahren das Spielfeld, so sind die Messwerte aufgrund der hohen Geschwindigkeit zu ungenau, liefern jedoch Informationen, wo sich die Tags ungefähr befinden. Beim Einziehen des Arms wird diese Information genutzt, um nur die wirklich interessanten Stellen langsam abzufahren (Abbildung 4.13, Variante c). Die restliche Strecke wird mit dem höchstmöglichen Tempo abgefahren. Durch den im Motor integrierten Winkelmesser hat sich die Genauigkeit dadurch nicht reduziert. Wie lange ein Scan-Vorgang dauert, hängt nun von der Anzahl und der Verteilung der Tags auf dem Spielfeld ab. Bei sieben Tags, die in kleinen Gruppen angeordnet sind, liegt der Zeitgewinn zum ursprünglichen Algorithmus bei ca. 30%.

Am meisten Zeit kann gewonnen werden, wenn eine ganze Zeile nicht gelesen werden muss. Da die Zeilen so nahe beieinander liegen, dass jeder Tag mindestens auf zwei Zeilen gelesen wird, besteht die Möglichkeit, einzelne Zeilen ganz zu überspringen (Abbildung 4.13, Variante d). Dies ist besonders bei dünn besiedelten Spielfeldern interessant. Um keine Verschiebung auf der Y-Achse zu erhalten muss jedoch sichergestellt werden, dass alle Zeilen in der Nähe eines Tags gelesen werden. Um dies zu garantieren, muss der Algorithmus auch bereits übersprungene Zeilen noch einlesen können. Diese Rückwärtsschritte können weiter optimiert werden, indem der Arm nicht ganz ausgefahren wird, sondern nur bis zu der Position, an welcher ein Tag erwartet wird. Der Zeitgewinn zum ursprünglichen Algorithmus liegt bei dieser Variante bei ca. 60%.

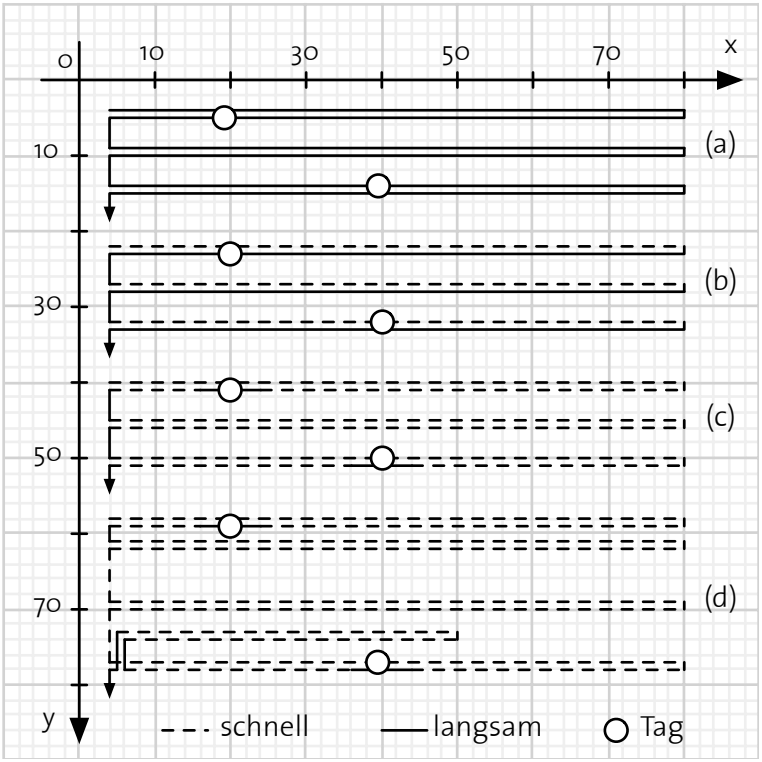


Abbildung 4.13: Bewegung beim Abtasten des Spielfelds.

Tabelle 4.2.: Verschiedene Algorithmen zum Abtasten des Spielfelds

Variante (Spielfeld: 40x40cm)	Zeit $[min]$	Abweichung $[mm]$
Variante a	11	2
Variante b	9	3
Variante c	7.5	3
Variante d	4.5	4
Variante d (kleinere Schrittweite)	6	2

Durch das kompliziertere Bewegungsmuster hat sich bei einem ersten Test die Genauigkeit auf $\pm 4mm$ verschlechtert. Durch ein Verkleinern der Schrittweite konnte das gut korrigiert werden, der Zeitgewinn hat sich jedoch auf ca. 45% reduziert. Tabelle 4.2 zeigt die erreichte Genauigkeit und die dazu benötigte Zeit für sämtliche angesprochenen Algorithmen.

Weitere Optimierungen sind ohne Änderungen an der Hardware kaum mehr möglich. So würde eine schnellere und präzisere Bewegung der Antenne und ein schnelleres RFID-Lesegerät die Zeit weiter reduzieren. Ein ganz anderer Ansatz wäre auch das optische Erkennen der Anwesenheit von Objekten auf dem Spielfeld. Über RFID müsste nur noch die ID bestimmt werden.

4.2.3. Messresultate

Um die optimale Geschwindigkeit für den Arm zu ermitteln, wurde eine Testreihe mit den verschiedenen Einstellungen des NXT-Moduls durchgeführt. Je nach Ausgangsleistung des Motors kann eine Geschwindigkeit von $0.7\frac{cm}{s}$ bis $11\frac{cm}{s}$ erreicht werden. Diese ist nicht linear zur Ausgangsleistung des Motors. Bei dieser Testreihe wurde nur die X-Achse beachtet.

Abbildung 4.14 zeigt die erreichten Geschwindigkeiten und den mittleren Messfehler beim Erkennen der Position von vier verschiedenen Tags. Die Resultate bei den einzelnen Tags variieren teilweise sehr stark, trotzdem zeigt der Mittelwert der Messfehler eine optimale Geschwindigkeit bei ungefähr $6\frac{cm}{s}$. Wird die Geschwindigkeit erhöht, steigt der Messfehler an. Eine niedrigere Geschwindigkeit benötigt zu viel Zeit für das Einlesen des Feldes, und

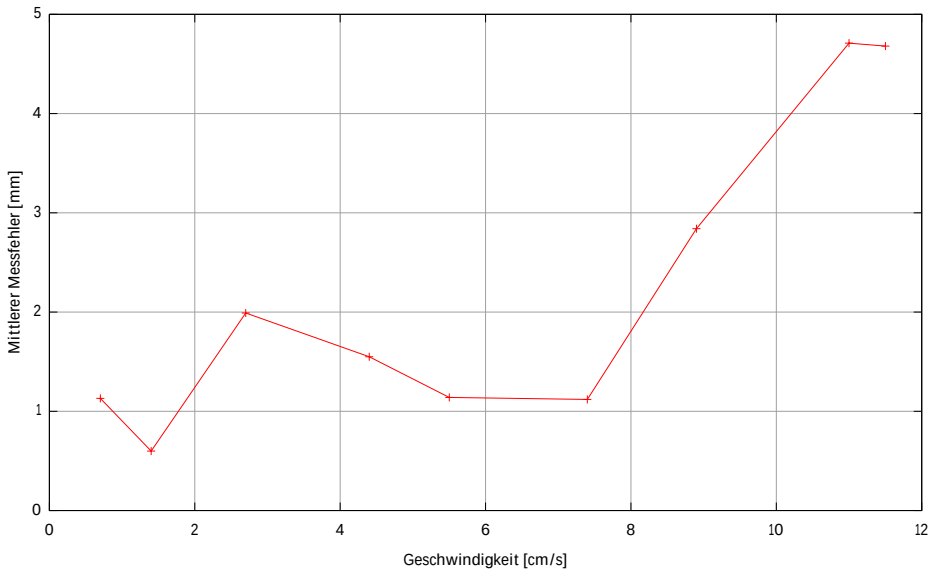


Abbildung 4.14: Messfehler bei unterschiedlicher Geschwindigkeit der Antenne.

der Motor beginnt zu schaukeln, sobald die Leistung des Motors zu gering ist. Die Messfehler sind in jedem Fall minimal, und die Genauigkeit liegt weit über den Erwartungen.

Das Ausmessen des ganzen Spielfelds war nicht so einfach wie erhofft. So haben sich die meisten Werte bei einer Wiederholung der Messreihe stark verändert. Vermutlich liegt die Ursache in den Kabeln, dem Ladezustand der NXT-Batterien und dem Spielfeld selbst. Die Kabel konnten nicht völlig frei geführt werden und sind immer wieder irgendwo hängen geblieben, was zu leichten Verschiebungen der Antenne führte. Lassen die Batterien im NXT-Modul nach, verlieren die Motoren an Kraft und können sich nicht mehr so genau positioniert werden. Dies hat sich besonders auf die Y-Achse ausgewirkt. Als besonders tückisch hat sich das leicht verzogene Raster auf dem Spielfeld herausgestellt. So wurde der Tag *D* aus Abbildung 4.15 nicht etwa falsch gelesen, sondern das Raster auf dem Spielfeld ist an diesem Punkt mehrere Millimeter verschoben. Auf dieses Problem wird in einem folgenden Abschnitt nochmals eingegangen.

Die Abbildung zeigt sieben verschiedene Tags, die jeweils in vier Durchgängen gelesen wurden. Dazu wurde der letzte der optimierten Algorithmen aus dem vorherigen Kapitel genutzt.

Die Verschiebungen auf der Y-Achse entstehen hauptsächlich durch die komplizierten Vor- und Rückwärtsbewegung bei diesem Algorithmus und können mit einfacheren Algorithmen reduziert werden. Es zeigt sich auch, dass der Mittelwert über alle Messwerte eines Tags noch näher am Mittelpunkt liegt.

Wie bereits erwähnt, sind die Verschiebungen dadurch zu erklären, dass die von Hand gemessenen Erwartungswerte zu ungenau waren. Da das ganze Spielfeld ca. 7 Zentimeter über dem Tisch auf einer nicht allzu robusten LEGO-Konstruktion montiert ist, war es kaum mehr möglich, die Tags von Hand zu vermessen, und jeglicher Versuch führte zu Messfehlern

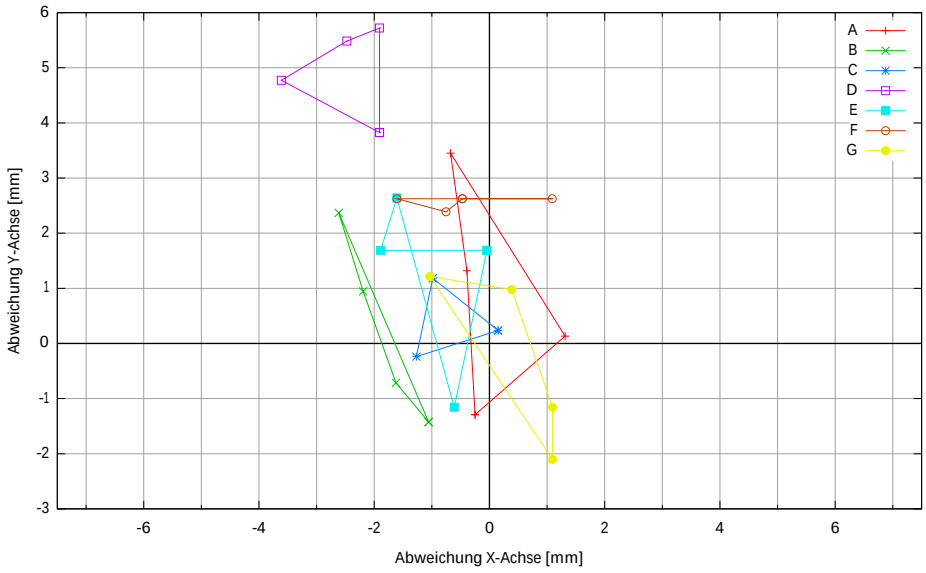


Abbildung 4.15: Gemessene Abweichung vom Soll-Wert (7 Tags, jeweils 4 Messungen).

von wenigen Millimetern. Die speziell grosse Abweichung beim Tag D konnte nach einigen Messversuchen eindeutig diesem Problem zugeschrieben werden. Auch die Tags B und F weisen eine starke Abweichung vom Nullpunkt auf. Ob diese Abweichungen von ca. 2 Millimetern auch nur Fehler der manuellen Vermessung sind, konnte jedoch weder bestätigt noch dementiert werden. Es fehlen Fixpunkte auf dem Spielfeld, die für eine manuelle Vermessung mit dieser Präzision unbedingt nötig wären. Für noch präzisere Abtastungen müsste das LEGO-Spielfeld durch eine andere, robustere Konstruktion ersetzt werden.

4.3. Koordinatentransformation

Die Koordinaten des RFID-Spielfelds entsprechen den Drehwinkeln der beiden Motoren, die des Simulators dem Koordinatensystem des Simulator-Fensters. Da in dem Spiel sämtliche Längenangaben in Zoll angegeben sind, wurde $\frac{1}{100}$ Zoll als Standardeinheit für das Spiel festgelegt und die Drehwinkel mussten entsprechend umgerechnet werden. Die Drehwinkel lassen sich kaum direkt in eine Längenangabe umrechnen, da sie zu stark von der Konstruktion des Spielfelds abhängig sind. Um auf eine aufwändige Kalibrierung zu verzichten, werden zwei zusätzliche Tags auf dem Spielfeld an klar vorgegebenen Positionen fest montiert. Durch diese beiden Referenzwerte können alle anderen Werte vom Eingabekoordinatensystem in das interne System transformiert werden.

Erst muss jedoch aus den vielen Sichtungen jedes einzelnen Tags der wahrscheinlichste Aufenthaltsort berechnet werden. Dazu dient die Klasse `Averager`, die nur über drei Methoden verfügt:

```
public void addSeeing(String tag, long x, long y);
public void addSeeings(String[] tags, long x, long y);
public void endOfRound();
```

Von der Datenquelle (Hardware oder Simulator) werden jeweils alle Sichtungen dieser Klasse übergeben. Die zweite Methode dient nur der einfacheren Nutzung, da alle Tags von einer Stelle gleichzeitig übergeben werden können. Wird ein Durchgang abgeschlossen, muss dies der Klasse über die dritte Funktion mitgeteilt werden. So kann sichergestellt werden, dass immer alle Werte übergeben werden. Die Klasse versucht jedoch Werte so früh als möglich weiterzuleiten. Ist die Klasse sicher, dass keine neuen Messungen für einen Tag eintreffen, wird der Mittelwert aller gemessenen Positionen pro Tag an die Koordinatentransformation weitergegeben.

Um die Koordinaten umrechnen zu können, werden erst die Faktoren m_x und m_y anhand der Referenztags bestimmt. Deren echte Position($phy_{x,y1,2}$) ist bekannt und mit den nun erhaltenen Messwerten($real_{x,y1,2}$) lässt sich folgende Berechnung durchführen.

$$\begin{aligned} m_x &= \frac{phy_{x1} - phy_{x0}}{real_{x1} - real_{x0}} \\ m_y &= \frac{phy_{y1} - phy_{y0}}{real_{y1} - real_{y0}} \end{aligned} \tag{4.2}$$

Sind diese Faktoren berechnet, können für alle weiteren Tags die echten Koordinaten mit der Formel 4.3 berechnet werden.

$$\begin{aligned} x &= \frac{x - phy_{x0}}{m_x} + real_{x0} \\ y &= \frac{y - phy_{y0}}{m_y} + real_{y0} \end{aligned} \tag{4.3}$$

Durch diese Umrechnung sind die Koordinatensysteme der Hardware (bzw. des Simulators) absolut unabhängig vom intern genutzen System. Ausserdem kann die Software in Zukunft problemlos mit anderen Spielfeldern genutzt werden, ohne dass die Umrechnung neu kalibriert werden muss. Einzige Voraussetzung sind zwei Referenztags, die an definierten Stellen im Spielfeld integriert sind.

Die eigentliche Implementierung der Koordinatentransformation besitzt zudem einige Besonderheiten. So werden alle gesichteten Tags zwischengespeichert, solange nicht beide Referenz-Tags gelesen wurden. Sind die Referenz-Tags bekannt, können alle anderen Werte berechnet und weitergeleitet werden. Da die Referenz-Tags bei jedem Durchgang erneut gescannt werden, können deren Koordinaten durch Mitteln der bisherigen Werte immer präziser festgestellt werden. Nimmt die Ladung der Batterien im NXT ab, so verschieben sich die Koordinaten leicht. Daher wurde kein üblicher Mittelwert berechnet, sondern jeweils die aktuellsten Werte stärker gewichtet. Details zum Vorgehen können dem Quellcode entnommen werden.

Die Schnittstelle zur Klasse `CoordTranslator`, die diese Umrechnung durchführt, ist simpel aufgebaut. So werden dem Konstruktor die IDs inklusive den dazugehörigen echten X- und Y-Koordinaten übergeben. Danach wird für jeden gelesenen Tag die Methode `tagSeen(...)` aufgerufen.

```
public CoordTranslator(String tag0Id, int tag0X, int tag0Y,
        String tag1Id, int tag1X, int tag1Y);

public void tagSeen(Tag tag);
```

4.4. Erkennen der Position und Ausrichtung von Objekten

Das Erkennen von Modellen mit einem Tag ist einfach, da die Koordinaten nach dem Mitteln und der Koordinatentransformation bereits der Position der Figur entsprechen. Bei grösseren Objekten mit mehreren Tags wird dies schwieriger. Das Programm muss wissen, in welchen Abständen zueinander welche Tags welches Objekt ergeben. Dies wurde durch Hinterlegen eines Prototyps für jeden Modell-Typ erreicht.

4.4.1. Algorithmus

In [8] wurde dieses Problem bereits gelöst. Der Algorithmus dreht den Prototypen jeweils $1°$, misst die Abweichung aller Tags des Prototyps zu den Messwerten und wiederholt diesen Vorgang 359 Mal für jeden Winkel zwischen $0°$ und $359°$.

Der in dieser Arbeit implementierte Algorithmus verfolgt einen anderen Ansatz. Für die gemessenen Punkte wird der Schwerpunkt (Mittelwert) berechnet. Ebenso beim Prototypen. Der Prototyp wird dann entsprechend verschoben, so dass beide Schwerpunkte aufeinander liegen (Abbildung 4.16, Bild a). Nun wird von jedem Punkt des Prototyps der Winkel zum gemessenen Gegenstück berechnet, wobei der Schwerpunkt als Zentrum genommen wird. So erhält man für jeden Tag einen Winkel. Der Mittelwert dieses Winkels entspricht der benötigten Rotation des Prototypen, um die gemessenen Punkte bestmöglich zu decken (Abbildung 4.16, Bild b).

Der Algorithmus funktioniert problemlos ab mindestens zwei Tags und auf das *Durchprobieren* aller Winkel konnte verzichtet werden. Selbst sehr grosse Abweichungen der Messwerte, was Messungenauigkeiten entspricht, werden gut verarbeitet und das Resultat ist präzise. Durch das gegenseitige Aufheben der Messfehler wird die Genauigkeit sogar noch erhöht (Tabelle 4.3). Man muss davon ausgehen, dass sich auch die Fehler beim manuellen Messen gegenseitig beeinflusst haben. Trotzdem - oder gerade deswegen - müssen die Werte kritisch betrachtet werden.

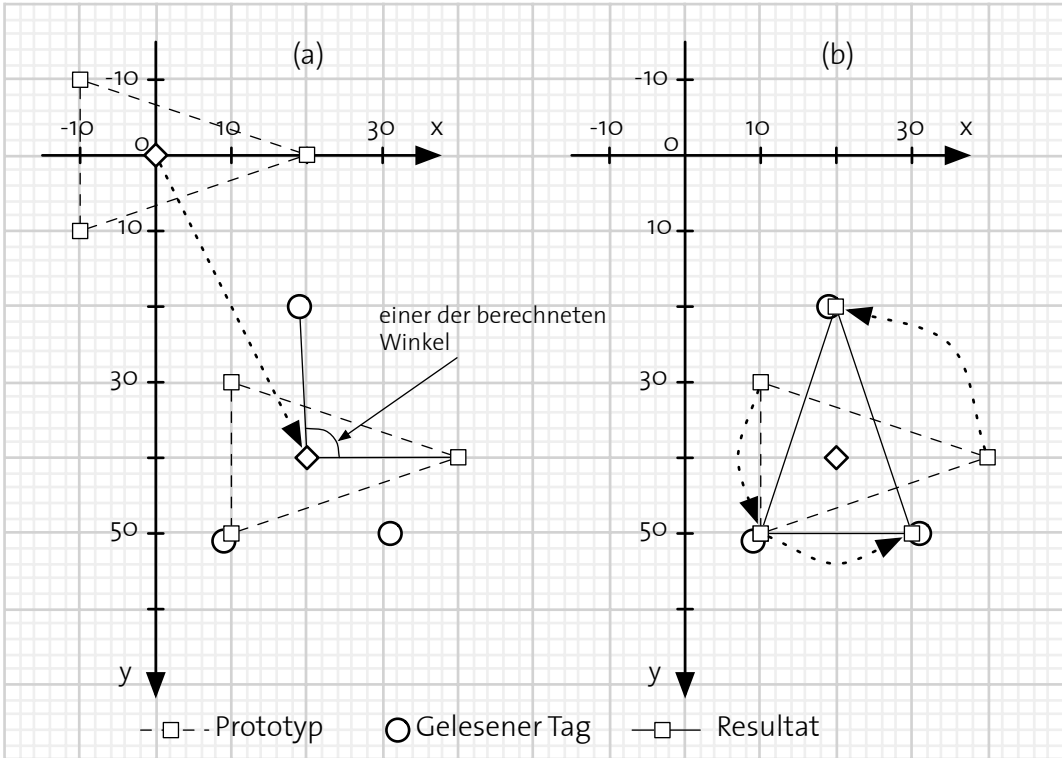


Abbildung 4.16: Erkennen der Position und Winkel von grösseren Objekten.

Bei der aktuellen Implementierung werden Messwerte für alle Tags des Prototyps benötigt, bevor das Objekt lokalisiert wird. Der Algorithmus lässt sich jedoch einfach verändern, um diesen Nachteil auszugleichen. Bei der Berechnung muss darauf geachtet werden, dass beim Prototypen nur die Tags für die Berechnung des Schwerpunktes genutzt werden, für welche auch Messwerte vorliegen. In der Praxis hat sich dies jedoch nie als Nachteil herausgestellt. Die Tags werden alle genügend oft gelesen. Solange diese sich nicht vom Objekt ablösen, funktioniert diese Variante einwandfrei.

Tabelle 4.3.: Genauigkeit beim Erkennen der Objektposition und Winkel

Objekt	Abweichung $[mm]$	Abweichung $[^\circ]$
einzelner Punkt	2	-
Dreieck	1.5	1.25
Viereck	0.5	1.00

4.4.2. Implementation

Die wichtigste Komponente ist die Klasse `Point`, die Methoden zum Berechnen von Mittelwerten und Winkeln besitzt und auch Punkte kreisförmig um andere Punkte verschieben kann. Die Klasse `Prototype` enthält den bereits angesprochenen Prototypen eines Objekts mit mehreren Tags.

Das Rotieren eines Punktes um eine vorgegebene Basis kann einfach gelöst werden, wenn erst der Punkt soweit verschoben wird, dass dieser auf dem Nullpunkt liegt. Die trigonometrischen Funktionen helfen beim Berechnen der neuen Koordinaten nach der Rotation.

```
public Point rotate(double degr, Point base) {
    double newX = (y-base.y)*Math.sin(degr)
            + (x-base.x)*Math.cos(degr) + base.x;
    double newY = (y-base.y)*Math.cos(degr)
            - (x-base.x)*Math.sin(degr) + base.y;
    return new Point((int)Math.round(newX), (int)Math.round(newY));
}
```

Das Berechnen eines Winkels zwischen zwei Punkten aus der Sicht eines weiteren, dritten Punkts lässt sich nicht über Winkelfunktionen wie $\sin$ und $\cos$ lösen. Liegen die beiden Punkte horizontal oder vertikal übereinander, können diese Funktionen keine sinnvollen Werte zurückgeben. Glücklicherweise besitzt Java eine Methode `Math.atan2(int, int)`, die X/Y-Koordinaten in Polarkoordinaten umrechnen kann, was die Aufgabe extrem vereinfachte. Sehr speziell an dieser Methode ist jedoch, dass als erstes Argument der Y-Wert und erst danach der X-Wert übergeben werden muss.

```
public double angleTo(Point trg) {
    return Math.atan2(trg.y - y, trg.x - x);
}
```

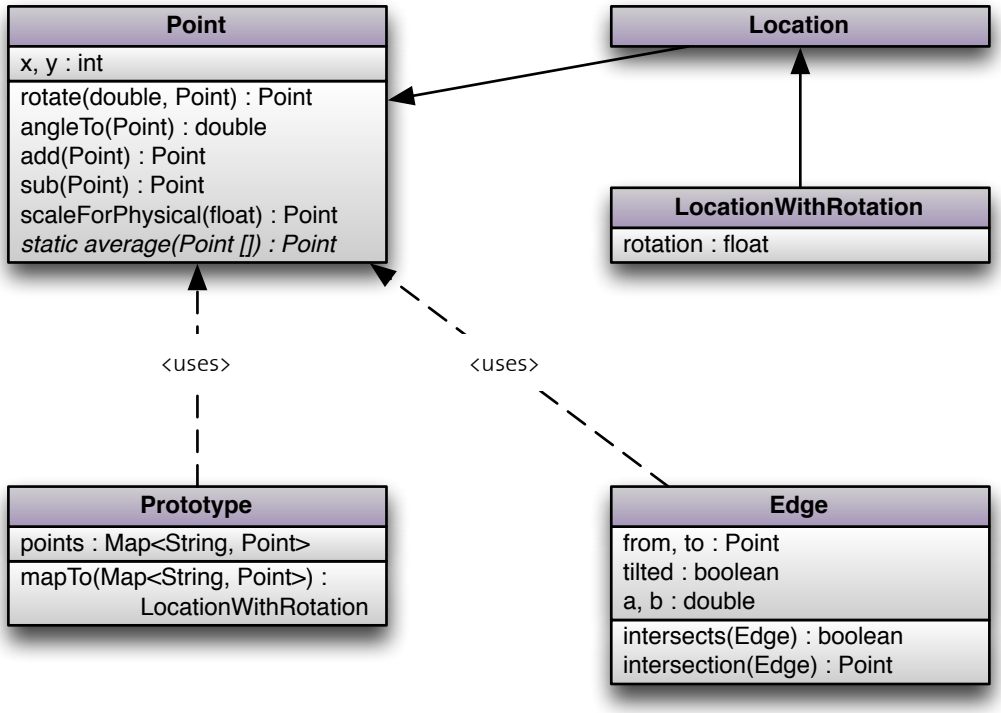


Abbildung 4.17: Klassendiagramm der Klassen zur Ortsbestimmung.

Als aufwändiger hat sich wie erwartet die Methode zum Berechnen des Mittelpunktes sowie der Drehung eines Objekts anhand der Messwerte und des Prototyps erwiesen. Auf die gar nicht so einfache Aufgabe des Berechnens eines Mittelwerts zweier Winkel wird im nächsten Abschnitt detaillierter eingegangen. Der folgende Algorithmus wurde bereits oben beschrieben und funktioniert in dieser Form einwandfrei.

```
public LocationWithRotation mapTo(Map<String, Point> real) {
    Map<String, Point> res = new HashMap<String, Point>();

    // calculate center of both figures
    Point protBase = Point.average(points.values().toArray(new Point[]{}));
    Point realBase = Point.average(real.values().toArray(new Point[]{}));
    Point delta = protBase.distanceTo(realBase);

    // for every tag, calculate angle betwenn prototype and measured value
    // and average the values
    float degr = (float)Math.PI*2;
    int mesu = 0;
    for (String id : points.keySet()) {
        res.put(id, points.get(id).add(delta));

        // calculate angle between corresponding points from protoype
        double div = realBase.angleTo(res.get(id))
                   - realBase.angleTo(real.get(id));

        // move to same base as existing results
        while (degr - div > 2*Math.PI/3) {
            div += 2*Math.PI;
        }

        // average with previous angles
        degr += (div - degr) / ++mesu;
    }

    degr %= Math.PI * 2;

    // rotate base of prototype to get correct base for object on gamefield
    Point nP = Point.ZERO.rotate(degr, protBase).add(delta);
    return new LocationWithRotation(nP.getX(), nP.getY(),degr);
}
```

4.4.3. Probleme beim Implementieren dieses Algorithmus

Die technische Umsetzung des Algorithmus hat sich als nicht ganz einfach erwiesen. Problematisch war nicht das Verschieben der Objekte oder das Drehen des Prototyps, sondern wider Erwarten die Berechnung des Mittelwerts über mehrere Winkel.

Aufgrund der besseren Verständlichkeit steht folgendes Beispiel in Grad anstatt in Bogenmass, obwohl im Programm jeweils mit Winkeln im Bogenmass gerechnet wird. Die Java-Funktion `double Math.atan2(int y, int x)` gibt Winkelwerte zwischen $-180°$ und $180°$ zurück. Liegt nun ein Winkel bei $-175°$ und der andere bei $175°$ lässt sich der Mittelwert nicht so einfach berechnen. Der übliche Ansatz $(a+b)/2$ führt zu einem Resultat von $0°$, richtig wäre $180°$. Auch das Addieren von $360°$ zu allen negativen Winkeln hilft nicht weiter. Bei obigem Beispiel erhält man dadurch $175°$ und $185°$, wo sich der Mittelwert gut berechnen lässt, aber das Problem ist nicht verschwunden sondern nur verschoben. Denn nun erhalten wir Paare wie $5°$ und $-5°$, was zu $355°$ wird. Hier lässt sich der Mittelwert auch nicht mehr korrekt berechnen.

Die Lösung muss also den immer vorhandenen Übergang dynamisch so verschieben, dass für die jeweils erhaltenen Winkel der Mittelwert berechnet werden kann.

Folgende Lösung hat sich als praxistauglich herausgestellt: Zu Beginn des Algorithmus wird von einem Winkel von $360°$ ausgegangen. Dieser wird mit dem ersten gemessenen Winkel verglichen. Ist der gemessene Winkel mehr als $200°$ kleiner als die erwarteten $360°$ werden $360°$ dazu addiert und dieser Wert als erster Mittelwert gespeichert. Für die weiteren berechneten Winkel wird nun dieses Vorgehen wiederholt. Ist der neue Winkel mehr als $200°$ kleiner als der bisher berechnete Mittelwert, werden $360°$ addiert und ein neuer Mittelwert wird über alle bisherigen Winkel berechnet. Es muss nur darauf geachtet werden, dass die zu Beginn angenommenen $360°$ nicht in das Endresultat einfliessen. Durch dieses Vorgehen gibt es keine feste Stelle mehr, an der ein Übergang im mathematischen Ring der Gradangaben die Berechnung stört.

Im Java-Programm wurden nicht alle Zwischenwerte gespeichert und stattdessen ein laufender Mittelwert eingeführt. Dieser wird über `avg` += (`angle` −`avg`)/ ++`counter` berechnet. Diese Berechnung hat auch den Vorteil, dass die zu Beginn angenommen $360°$ beim ersten Schritt automatisch wegfallen, da der Counter noch auf Null steht.

4.5. Datenmodell

Die aktuellen Koordinaten werden von der Klasse `TagInterpreter` den jeweiligen Spielobjekten zusammen mit der ID des Tags übergeben. Die Spielobjekte berechnen daraus die Position mit dem oben erwähnten Algorithmus und informieren die Spiellogik über den neuen Zustand. Die Klassen der Spielobjekte sind wie in Abbildung 4.18 gezeigt implementiert.

4.5.1. Verwendete Klassen

Die gemeinsame Basisklasse `GameObject` wurde gewählt, da bei einigen Ereignissen, wie beispielsweise dem Würfeln, jede Art von Objekt mitgegeben werden kann. Für das Erkennen von Objekte, die auf einer Linie liegen, wurde `VisibleGameObject` eingeführt. Die Klasse `AbstractGameModel` enthält einige Methoden, die für alle Modelle auf dem Spielfeld gelten. Die beiden abgeleiteten Klassen `VehicleGameModel` und `NonvehicleGameModel` entsprechen den Spielerobjekten auf dem Spielfeld. Bei Fahrzeugen werden *montierte* Waffen verwendet. Diese besitzen eine Position innerhalb des Objekts und einen möglichen Winkelbereich, aus dem geschossen werden kann. Beliebige Kombinationen aus Modellen eines Spielers werden zu einer Einheit, der `GameUnit`, zusammengeschlossen. Die meisten Aktionen innerhalb des Spiels werden über die Einheit, nicht über einzelne Modelle, abgewickelt.

Die Klassen besitzen kaum eigene Intelligenz. Sie sind überwiegend zur Speicherung von Daten gedacht. Nur wenige Methoden wie zum Beispiel `intersectsWith(Edge)` in `VisibleGameModel` oder die Methode `reaches(AbstractGameModel)` der Klasse `GameWeapon` sind implementiert. Die meisten Methoden befindet sich in der Spiellogik.

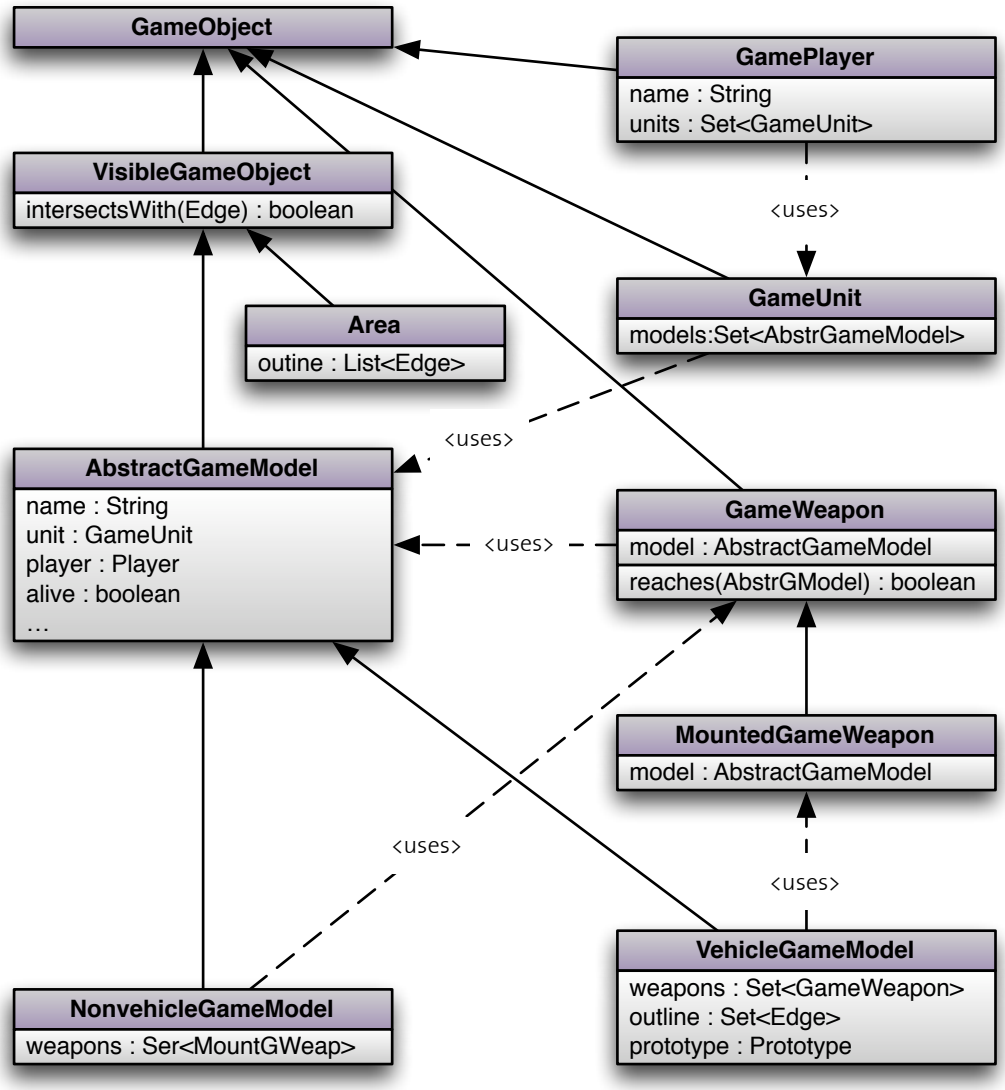


Abbildung 4.18: Klassendiagramm der Spielobjekte.

4.5.2. Laden der Figuren

Sämtliche Attribute der Spielfiguren sind in einigen XML-Dateien abgelegt. Im Anhang C werden die einzelnen Dateien detailliert beschrieben.

Der bereits kurz erwähnte `TagInterpreter` hat unter anderem die Aufgabe, benötigte Spielfiguren vorzubereiten. Dazu ist die Klasse auf drei Konfigurationsdateien angewiesen.

`modelTemplates.xml` enthält für jede Art Modell eine Vorlage. Nebst dem Wert der Einheit sind darin weitere vom Spiel benötigte Attribute als auch die Waffen vermerkt. Es wird

jedoch noch keine Tag-ID angegeben. Diese Datei dient nur zum Beschreiben der verschiedenen Arten von Modellen.

`concreteUnits.xml` enthält die eigentlichen Einheiten. Zu jeder Einheit ist der Spieler, dem diese Einheit gehört, gespeichert, und dazu deren sämtliche Modelle. Bei den einzelnen Modellen wird auch die Tag-ID angegeben. So kann der `TagInterpreter` jeweils die komplette Einheit laden, sobald eines der Modelle auf dem Spielfeld erkannt wird.

Die Eigenschaften der Waffen sind in der Datei `weaponTemplates.xml` festgelegt. Auf diese Konfiguration wird in `modelTemplates.xml` verwiesen. Anhand dieser drei Konfigurationsdateien kann der `TagInterpreter` neue Instanzen der Spielobjekte erzeugen und bei der Logik-Klasse registrieren, sobald diese gebraucht werden. Wichtig ist, dass sämtliche Einheiten und Modelle vor dem Start des Programms in diesen Dateien konfiguriert worden sind. Im Kapitel 5.2 wird eine bessere Alternative kurz vorgestellt.

4.6. Benutzerinteraktion

Ein wichtiger Teil der Benutzerinteraktion findet indirekt über das physikalische Spielfeld statt. Im aktuellen Entwicklungsstand des Prototypen war es leider nicht möglich, sämtliche Interaktion über dieses Spielfeld zu lösen, und die Spieler müssen mit der Maus und dem PC interagieren. Dabei muss zwischen drei Arten der Interaktion unterschieden werden.

Einerseits gibt es das Würfeln, das bei diesem Spiel eine sehr wichtige Rolle spielt. Es muss häufig, aber auch mit einer grossen Anzahl von Würfeln gewürfelt werden. Diese Art der Interaktion könnte auch über das Spielfeld stattfinden, entweder über RFID [19] oder über eine optische Erkennung der Punkte auf den Würfeln. In der aktuellen Version der Anwen-

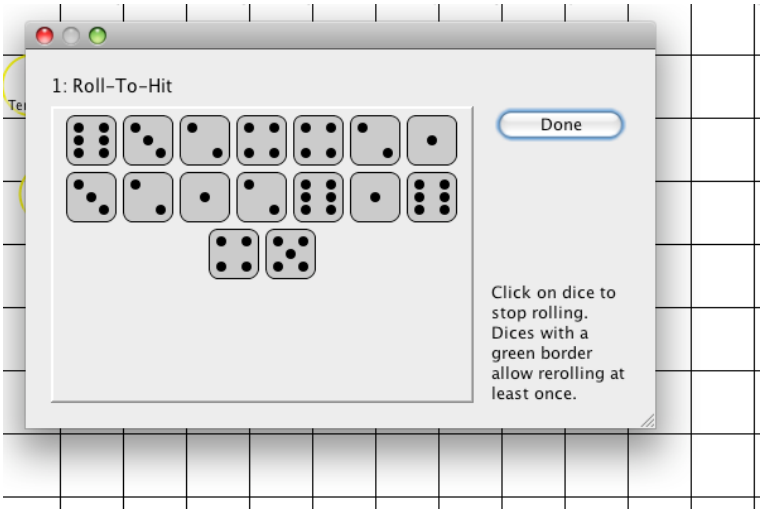


Abbildung 4.19: Würfel-Simulator.

dung werden die Würfel simuliert, da die vorhandene Technik keine genügend grosse Anzahl Würfel zulässt. Abbildung 4.19 zeigt den Würfel-Simulator.

Eine andere Art der Interaktion ist die Information der Spieler. Die Resultate der Spiellogik müssen übermittelt werden, ohne die Spieler zu sehr abzulenken. Häufig gibt es auch Hinweise, die nur sehr kurzfristig gültig sind, wie zum Beispiel die Resultate des Würfels. Dazu musste ein Weg gefunden, werden um kurz- und langlebige Informationen zu übermitteln.

Leider konnte auch nicht darauf verzichtet werden, von den Spielern zusätzliche Informationen zum Spielgeschehen abzufragen. Im aktuellen Zustand muss das System wissen, wann gewürfelt werden muss, um den entsprechenden Simulator anzuzeigen. Aber auch die Regeln sind teilweise von den Entscheiden der Spieler abhängig, die das System mit den bisherigen Mitteln nicht selbstständig erkennen kann. Das ganze Spielgeschehen müsste in einem nichtdeterministischen Automaten abgebildet werden, der nach und nach aufgrund der Informationen vom Spielfeld den jeweils aktuell gültigen Zustand evaluieren kann.

Die ersten beiden Arten wurden in eine gemeinsame Interaktionsebene ausgelagert, da sich diese beiden Arten gegenseitig ergänzen. Das Würfeln führt zu Resultaten, die wieder ange-

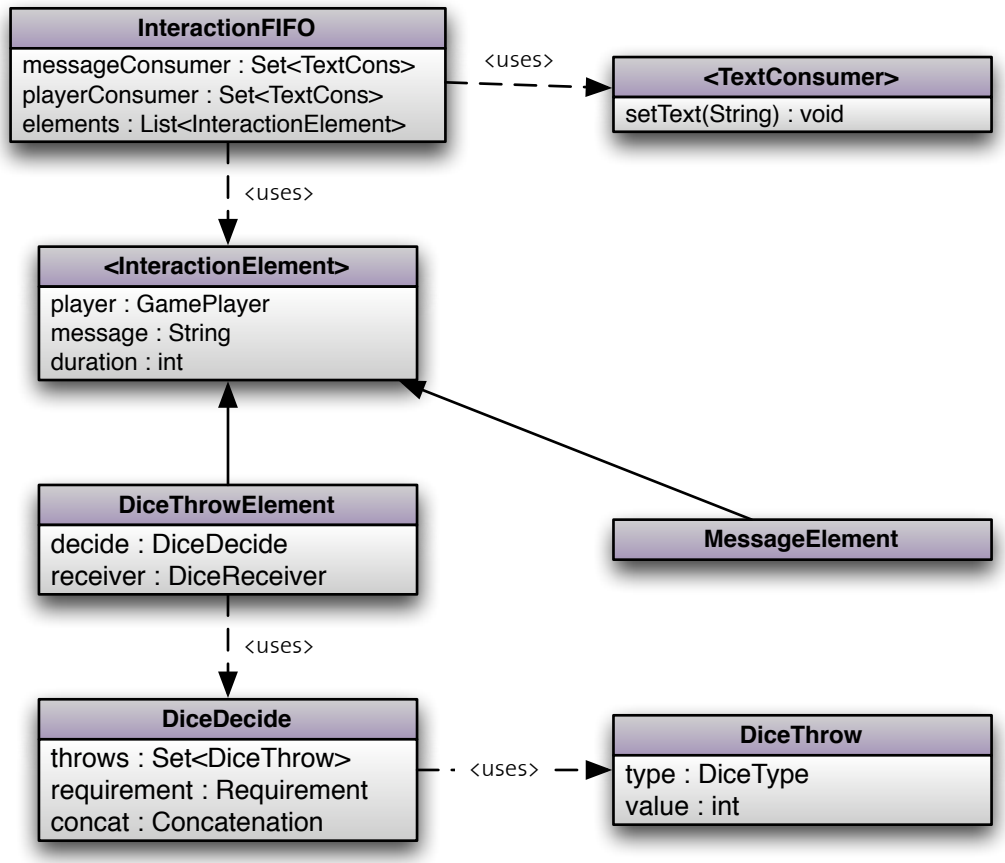


Abbildung 4.20: Klassendiagramm der Interaktionsklassen.

zeigt werden müssen, und die Informationen müssen angezeigt werden, bevor das Würfeln beginnt. Um dies zu ermöglichen, wurde eine Art Pipeline für diese Art der Interaktion eingeführt. Meldungen und Würfelereignisse können registriert werden, und die Pipeline zeigt die Meldungen und Würfelsimulatoren in der korrekten Reihenfolge an und überprüft, dass alles genügend lange auf dem Bildschirm gezeigt wird. Abbildung 4.20 zeigt die beteiligten Klassen.

Die `InteractionFIFO` stellt die Pipeline dar, die die Meldungen zwischenspeichert und später anzeigt. Dabei werden die Elemente `diceThrowElement` und `MessageElement` unterstützt. während das zweite nur einen Text auf dem Bildschirm ausgibt, kann das erste den Würfelsimulator starten. Über die Hilfsklassen `DiceDecide` und `DiceThrow` wird dieser konfiguriert. Die Resultate werden an einen bereits beim Erstellen des Würfel-Elements festgelegten Empfänger gesendet.

Die letzte Art der Interaktion, die Hinweise zum Spielablauf, könnten direkt in das auf dem Bildschirm angezeigten Spielfeld integriert werden. So muss häufig ein Modell oder eine Einheit ausgewählt werden. Dies kann mit einem Klick auf die entsprechenden Objekte gemacht werden. Könnte das Spielfeld schneller gescannt werden, könnte man dies mit einem speziellen, mit RFID-Tag markierten Marker, lösen. Diese Marker, die zum Beispiel in Form einer kleinen Fahne auftreten könnten, sollten jedoch in wenigen Sekunden erkennbar sein, um den Spielfluss nicht unnötig aufzuhalten. Die bisherigen Zeiten von mehreren Minuten sind dafür ungeeignet.

Das Abfragen von Informationen zu einem Modell konnte gut über das virtuelle Spielfeld gelöst werden. Fährt ein Spieler mit der Maus auf ein Modell, werden sämtliche Informationen in einem Tool-Tip-Feld angezeigt. Abbildung 4.21 gibt ein Beispiel. Die feinen Linien zwischen den einzelnen Modellen zeigen, welche Modelle zu einer Einheit gehören, während die dickere Linie die Bewegung eines Modells in dieser Runde anzeigt. Der kleine schwarze Kreis markiert die Position am Anfang der Runde, das Modell die aktuelle Position.

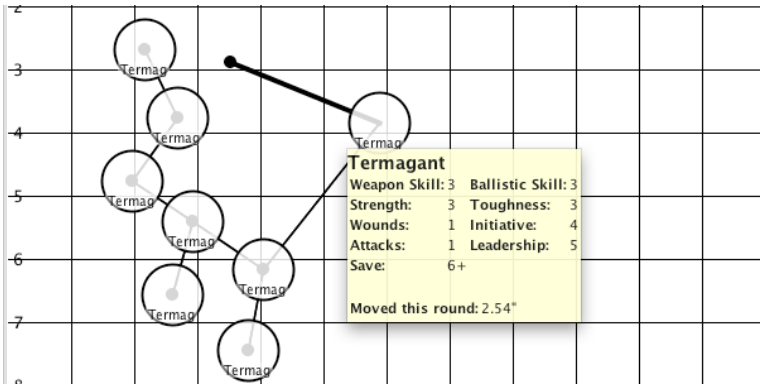


Abbildung 4.21: Darstellung der Informationen zu einer Einheit.

Wird eine Figur zu weit bewegt, färbt sich die Linie, welche den zurückgelegten Weg anzeigt, rot. Ein zusätzlicher Hinweis unter dem Spielfeld dient als weitere Information für die Spieler.

Eine feine rote Linie zwischen zwei Modellen zeigt, dass die Distanz zu gross und die Einheit daher nicht mehr zusammenhängend ist. Dies ist auf Abbildung 4.22 ersichtlich.

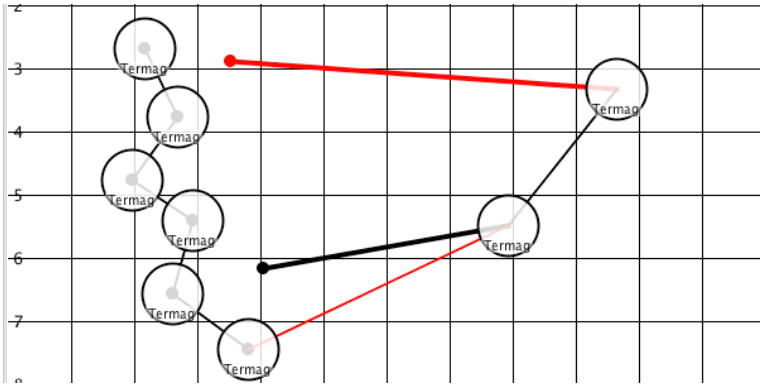


Abbildung 4.22: Die Darstellung unerlaubter Bewegungen.

Abschliessend lässt sich sagen, dass die aktuelle Implementierung bei weitem nicht optimal ist. Der Spieler fokussiert zu stark auf Bildschirm und Maus. Dies sollte noch besser untersucht werden.

4.7. Grafische Umsetzung

Auch wenn der Schwerpunkt nicht bei grafischen Effekten liegt, ist die Darstellung des virtuellen Spielfelds ein wichtiger Punkt für den Spieler. Die Figuren, die Einheitszusammengehörigkeit und die Informationen zu den Modellen müssen deutlich sichtbar sein. Abbildungen 4.21 und 4.22 zeigten bereits Ausschnitte aus dem virtuellen Spielfeld.

4.7.1. Aufbau des Programmfensters

Abbildung 4.23 zeigt das vollständige Programmfenster auf dem PC. Das Fenster wurde absichtlich schlicht und übersichtlich gehalten, da der Computer keine wichtige Rolle während des Spiels haben darf. Der grösste Bereich des Fensters zeigt das virtuelle Spielfeld. Unter dem Spielfeld befindet sich die Statuszeile. Auf dieser werden die im letzten Kapitel erwähnten Informationen an die Spieler ausgegeben. Auf der rechten Seite befindet sich meistens nur eine einzige Schaltfläche, die dem Spieler erlaubt, in die nächste Phase des Spiels zu wechseln. Das Bildschirmfoto zeigt die Vorbereitung vor dem eigentlichen Spiel. Hier wird auch für beide Spieler die Stärke der jeweils aktuellen Armee angezeigt.

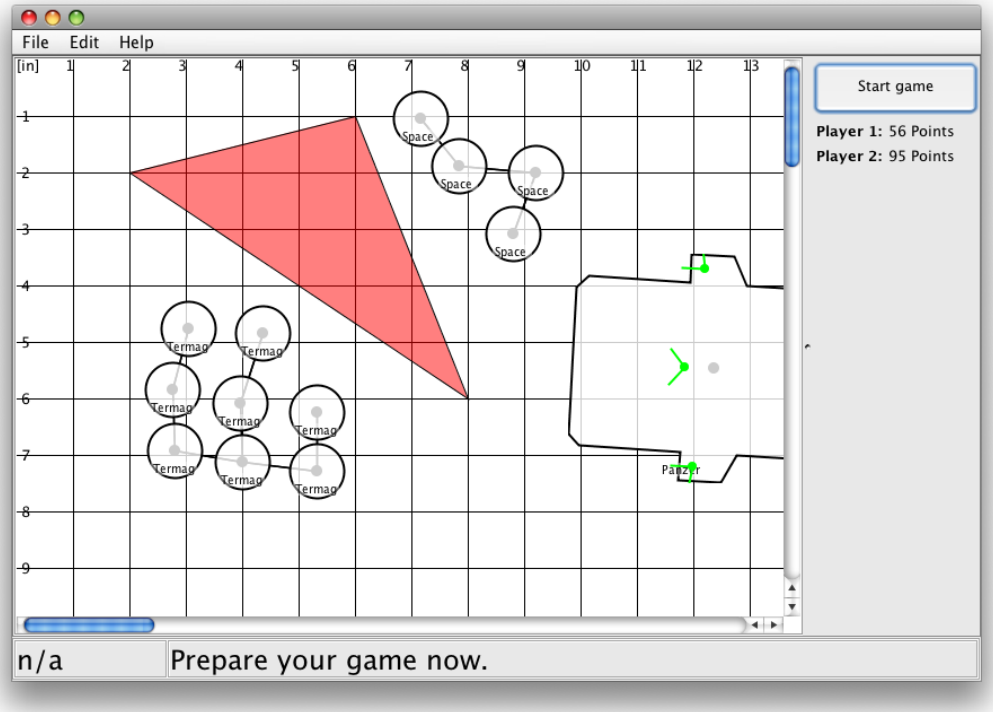


Abbildung 4.23: Das virtuelle Spielfeld während der Vorbereitungsphase.

4.7.2. Implementation

Jeder Spielfigur wurde ein Objekt der Klasse `ModelComponent` zugewiesen. Diese von `JComponent` abgeleitete Klasse ist für das Zeichnen der Figur zuständig. Bei einfachen Modellen ist diese Komponente nur so gross, dass der Kreis, welcher die Figur markiert, darin gezeichnet werden kann. Wie man beim Fahrzeug im rechten Teil des Bilds gut erkennen kann, muss die Komponente für Fahrzeuge wesentlich grösser sein. Aber auch diese sind genau so gross wie die Aussenkante des Fahrzeugs. Durch diese knapp eingepassten Komponenten kann ein Mausklick auf ein Modell sehr einfach erfasst werden. Ausserdem kann sich ein Modell - wenn auch mit der Hilfe der `ModelComponent`-Klasse - selbstständig zeichnen, was zukünftige Erweiterungen vereinfacht. Abbildung 4.24 zeigt die Zusammenhänge.

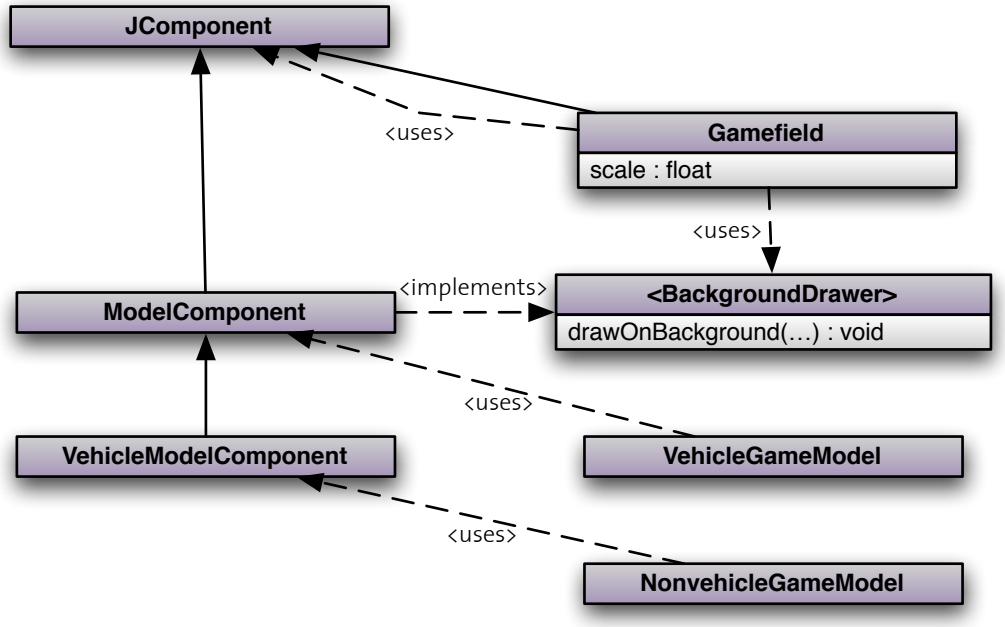


Abbildung 4.24: Klassendiagramm der grafischen Elemente.

Das Spielfeld ist auch von der Klasse `JComponent` abgeleitet und wird als Container für die enthaltenen Figuren genutzt. Das Spielfeld zeichnet nach dem Raster im Hintergrund erst die speziellen Gebiete, die Verbindungslinien zwischen den Figuren und die dickeren Linien, die den zurückgelegten Weg eines Modells darstellen. Da das Spielfeld selbst die dazu nötigen Informationen nicht besitzt, wird eine spezielle Callback-Funktion der Klasse `ModelComponent` verwendet. Durch diese Methode kann eine Figur ausserhalb der eigenen Komponente auf den Hintergrund zeichnen.

```
public void drawOnBackgroud(Graphics2D g2);
```

Erst wenn der Hintergrund vollständig gezeichnet wurde, werden die normalen `paint`-Methoden der einzelnen Figuren aufgerufen. Die Gebiete auf dem Spielfeld wurden nicht über eigene Komponenten gelöst, da mit diesen nie interagiert werden muss. Die Klasse `GameLogic` kennt die Gebiete und sorgt selbst für deren Darstellung. Dies führte zu folgender Implementierung des Spielfelds:

```
@Override
public void paint(Graphics g) {

    // draw grid
    g.drawString("[in]", 0, 10);
    int scaledMaxY = (int) (MAX_HEIGHT * scale);
    for (int x = X_GRID_STEP; x < MAX_WIDTH; x += X_GRID_STEP) {
        int scaledX = (int) (x * scale);
        g.drawLine(scaledX, 0, scaledX, scaledMaxY);
        g.drawString(Integer.toString(x / X_GRID_STEP), scaledX - 7, 10);
    }
```

```
    int scaledMaxX = (int) (MAX_WIDTH * scale);
    for (int y = Y_GRID_STEP; y < MAX_HEIGHT; y += Y_GRID_STEP) {
        int scaledY = (int) (y * scale);
        g.drawLine(0, scaledY, scaledMaxX, scaledY);
        g.drawString(Integer.toString(y / Y_GRID_STEP), 4, scaledY + 5);
    }

    // enable antialiasing
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    // let GameLogic draw on background (for areas)
   GameLogic.getInstance().drawBackground(g2);

    // let Components draw on background
    for (Component component : getComponents()) {
        if (component instanceof BackgroundDrawer) {
            ((BackgroundDrawer) component).drawOnBackgroud(g2);
        }
    }

    // draw Components
    super.paint(g);
}
```

In obigem Quelltext sieht man das Klassenattribut `scale`. Sämtliche Zeichenoperationen müssen für die Bildschirmausgabe skaliert werden, da das Programm intern $\frac{1}{100}$ Zoll als Einheit verwendet. Das Attribut `scale` ist bisher auf 0.5 fixiert, könnte jedoch einfach geändert werden, um ein Skalieren des virtuellen Spielfelds zuzulassen.

In der Entwicklung des Programms wurde sehr viel Wert auf die Unabhängigkeit der verschiedenen Koordinatensysteme gelegt. Die von der Hardware oder dem Simulator gelieferten Werte transformiert der Koordinatentransformator in das interne System, und die Ausgabe wird vom Spielfeld nochmals skaliert, um eine optimale Darstellung zu ermöglichen.

Fahrzeuge auf dem virtuellen Spielfeld

Die einzelnen Punkte der Aussenlinie des Fahrzeugs wurden aus der XML-Datei gelesen. Da die Fahrzeuge nicht wie die einfachen Modelle symmetrisch aufgebaut sind, muss das ganze Modell gedreht und skaliert werden, um korrekt auf dem Bildschirm dargestellt zu werden.

Das eigentliche Fahrzeug wird dabei innerhalb der grafischen Komponente gezeichnet. Um dies zu erreichen, berechnet das Programm zuerst die am weitesten links, rechts, oben und unten liegenden Punkte der Aussenlinie des Fahrzeugs. Die Komponente wird daraufhin verschoben und die Grösse angepasst, bevor die Aussenlinie des Fahrzeugs gezeichnet wird.

Da innerhalb dieser Komponente ein anderes Koordinatensystem gilt als auf dem Spielfeld, entstanden einige Komplikationen. Sehr schnell verliert man bei den verschiedenen Systemen und dem Drehen des Objekts die Übersicht. Besonders schwierig war das Einzeichnen der auf dem Fahrzeug montierten Waffen. Diese sind an einem Punkt auf dem Fahrzeug angebracht und besitzen nur einen gewissen Winkel, in den mit dieser Waffe geschossen werden kann.

Auf Abbildung 4.23 sind die drei Waffen des Fahrzeugs ersichtlich. Jede Waffe besitzt grüne Markierungen, die den erlaubten Schusswinkel anzeigen.

4.8. Spiellogik

Die Spiellogik wurde unterteilt in verschiedene Module, die die verschiedenen Phasen des Spielablaufs abdecken. So konnte die Komplexität des Spiels ein wenig reduziert werden. Für den Prototypen wurde jedoch bei weitem nicht der komplette Regelsatz abgedeckt. Grundlegendes Spielen ist möglich, jedoch sind Fahrzeuge nicht funktionstüchtig und spezielle Eigenschaften von Modellen können nicht genutzt werden.

Trotz dieser Reduktion auf grundlegende Regeln enthalten die Logik-Klassen viel Code und wurden relativ komplex. Die Spielregeln liessen sich am einfachsten mit einem endlichen Automaten beschreiben, von Java wird dies aber nicht unterstützt. Das Umschreiben in objektorientierten Code führte zu einer unübersichtlichen Abfolge der Methodenaufrufe.

4.8.1. Module der Spiellogik

Die Klasse `GameLogic` ist hauptsächlich ein Adapter, der jeweils die aktuelle Spielphase kapselt. Zusätzlich enthält die Klasse Informationen zu den Gebieten auf dem Spielfeld und kann daher auch eine Liste aller Objekte auf einer Linie zurückgeben. Dies wird hauptsächlich bei der Berechnung des Sichtkontakts genutzt. In den folgenden Abschnitten werden die einzelnen Module beschrieben.

Die `BuildupLogic` wird während der Vorbereitung, dem Setzen der Figuren auf dem Spielfeld, genutzt. Um ausgeglichene Armeestärken zu erhalten, werden jeweils für alle beteiligten Spieler die Punkte der im Spiel befindlichen Einheiten addiert und in der Seitenleiste angezeigt. Da das Programm nicht erkennen kann, wann der Aufbau abgeschlossen ist, muss dies über eine Schaltfläche bestätigt werden. Hier könnte ein mit RFID bestückter Marker die Schaltfläche ersetzen.

Nun beginnt die erste Phase des Spiels. Der erste Spieler kann seine Figuren bewegen. Die entsprechenden Regeln sind in der `MoveLogic` abgedeckt. Dazu gehören das Überwachen der zurückgelegten Distanz, der neuen Position, aber auch das Starten des Würfel-Simulators, falls der Spieler gefährliches oder schwieriges Terrain passiert hat. Auch diese Spielphase muss durch einen Klick auf eine Schaltfläche manuell beendet werden. Es wäre jedoch möglich den Übergang automatisch zu erkennen. Der Vorteil der Schaltfläche liegt einerseits darin, dass diese nicht fälschlicherweise ausgelöst wird, andererseits werden dadurch die Phasen klar getrennt.

Die `ShootLogic` kommt nicht ohne Interaktion über das auf dem Bildschirm gezeichnete Spielfeld und einer Maus aus. Dies liegt in erster Linie daran, dass der Würfel-Simulator gestartet werden muss. So wählt der Spieler seine Schützen sowie das Ziel aus. Die nun folgenden Würfelvorgänge werden auf dem PC ausgeführt, der automatisch die Resultate bestimmt.

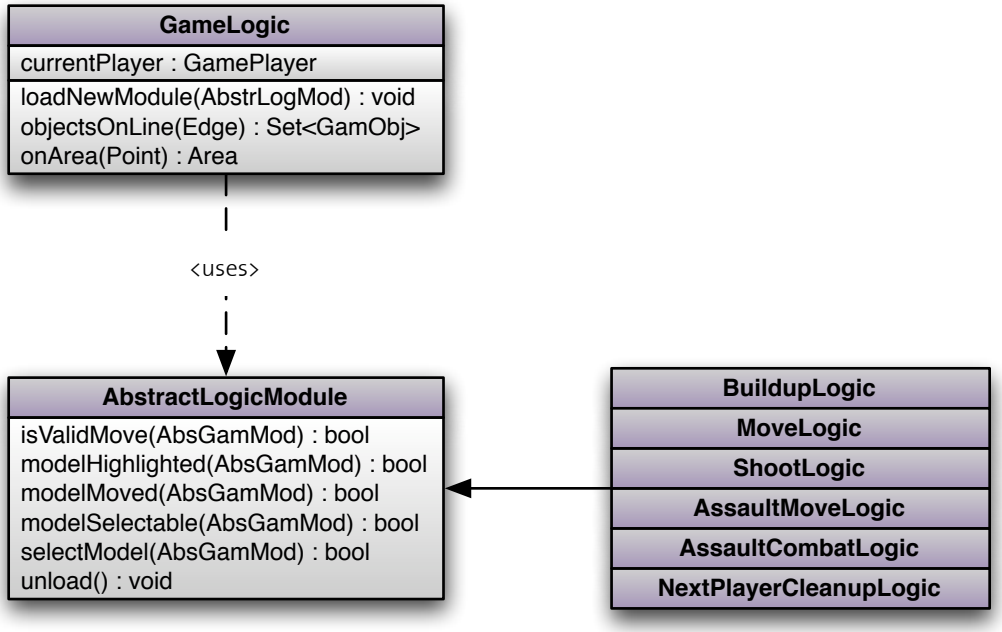


Abbildung 4.25: Klassendiagramm der Spiellogik.

Diese Phase könnte mit Markern und RFID-Würfeln [19] wesentlich interessanter gestaltet werden. Auch diese Spielphase wird mit einem Klick auf die Schaltfläche abgeschlossen.

Die nun folgende Nahkampf-Phase wurde in zwei Module aufgeteilt. Ein Erstes überwacht die Bewegungen während dieser Phase, das Zweite übernimmt die Auflösung des eigentlichen Kampfgeschehens. Die Bewegungsphase kommt ohne Interaktion mit dem PC aus, bei der Kampfphase ist dies nicht möglich, da dem Spieler die Wahl gelassen werden muss, welche Kämpfe in welcher Reihenfolge stattfinden. Auch muss der Würfelsimulator entsprechend konfiguriert und gestartet werden.

Die `NextPlayerCleanupLogic` setzt die nur für eine Runde gültigen Attribute in den Spielobjekten zurück und aktiviert den nächsten Spieler, falls das Spiel nicht von jemandem gewonnen wurde. Als Endbedingung für das Spiel wird nur die totale Vernichtung der Gegner in diesem Prototypen erkannt.

4.8.2. Erkennen des Sichtkontakts

Um festzustellen, ob zwei Objekte Sichtkontakt haben und somit aufeinander schiessen können, müssen alle Objekte zwischen diesen erkannt werden. Da das Spiel kein komplettes 3D-Modull nutzt, reichen 2D-Koordinaten und die Positionen aller Spielobjekte.

Auf dem Spielfeld auf dem Computer sind die Objekte für einen Menschen optisch sehr schnell erfassbar. Für den Computer musste ein entsprechender Algorithmus gesucht werden, der erkennen kann, ob Flächen, Fahrzeuge oder Gelände eine vorgegebene Linie schneiden.

Fahrzeuge und Gelände werden dazu auf die Aussenbegrenzung der Figuren, bzw. Flächen reduziert. Schneidet sie die Sichtlinie eine der Linien, die das Objekt begrenzen, so muss das Objekt auf dieser Linie sein. Die kleineren Modelle haben jeweils einen runden Fuss und benötigen daher einen anderen Ansatz. Da die Sichtlinie jeweils bekannt ist, wurde durch den Mittelpunkt der kleinen Modelle eine Linie gelegt, die senkrecht zur Sichtlinie steht und deren Länge dem Durchmesser der Grundfläche der Einheit entspricht. So lassen sich auch diese Modelle auf eine einzelne Linie reduzieren und können identisch geprüft werden wie die anderen Objekte. Die Abbildung 4.26 zeigt ein Beispiel.

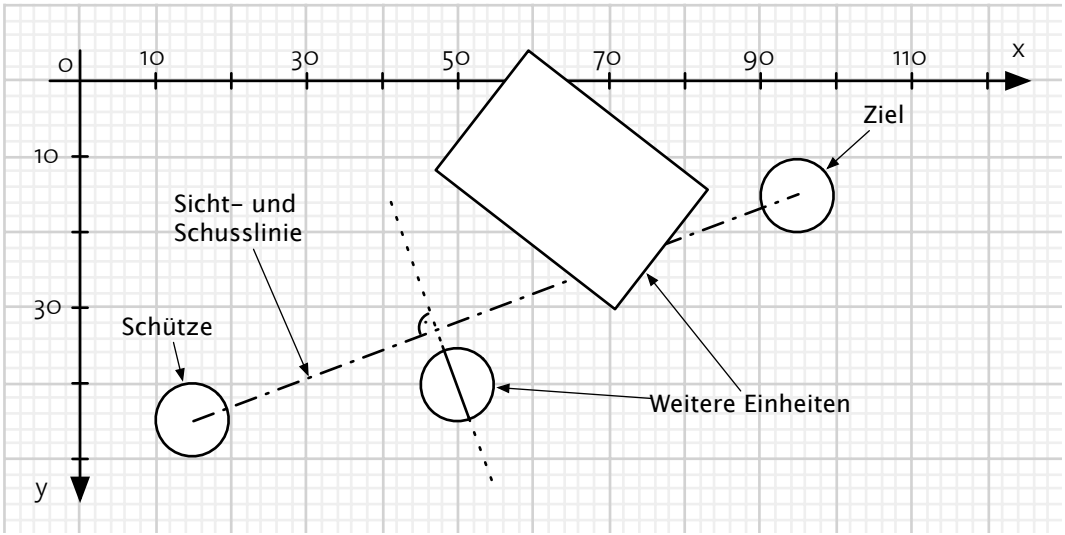


Abbildung 4.26: Grafische Darstellung der Sichtlinie.

Somit konnte das Problem auf das Erkennen eines Schnittpunkts zwischen zwei Linien reduziert werden. Die Linien bestehen jeweils aus zwei Koordinaten, die Start- und Endpunkt markieren. Der naheliegendste Ansatz besteht darin, beide Linien als Geradengleichung der Form $y = a * x + b$ darzustellen und danach die beiden Gleichungen zu kombinieren. Durch die beiden Variablen a und b können die meisten Geraden dargestellt werden. Nicht jedoch vertikale Linien. Bei diesen wird $a = \infty$ und b lässt sich überhaupt nicht mehr berechnen. Auch Linien, die nicht ganz vertikal sind lassen sich kaum darstellen, da beide Variablen gegen $\pm\infty$ gehen.

Als Lösung wurde für Geraden, die eher vertikal als horizontal sind, das Gleichungssystem gekippt. Anstatt $y = a*x+b$ wurde $x = a*y+b$ berechnet. Diese Gleichung kann alle Linien bis auf die Horizontalen darstellen. Dieses *Kippen* ermöglicht die Darstellung der Geraden, erschwert jedoch die Berechnung des Schnittpunkts durch das Einführen von Spezialfällen.

Ob eine Linie eher horizontal oder eher vertikal liegt, lässt sich über die Ungleichung $|X_2 - X_1| \leq |Y_2 - Y_1|$ testen. Ist diese erfüllt, ist die Linie eher horizontal, ansonsten verti-

kal. Ist entschieden, wie die Geradengleichung aussehen soll, kann diese berechnet werden. a_H, b_H stehen für horizontale Geraden, a_V, b_V für vertikale Geraden.

$$a_H = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad a_V = \frac{X_2 - X_1}{Y_2 - Y_1}$$
$$b_H = Y_1 - (a * X_1) \qquad b_V = X_1 - (a * Y_1)$$

Aus diesen Formeln lässt sich folgende Methode ableiten. Diese entscheidet anhand der übergebenen Punkte, ob es sich um eine eher horizontale oder vertikale Linie handelt und berechnet die entsprechende Geradengleichung.

```
public Edge(Point from, Point to) {
    this.from = from;
    this.to = to;

    int dX = to.getX()-from.getX();
    int dY = to.getY()-from.getY();

    // horizontal line
    if (Math.abs(dX) >= Math.abs(dY)) {
        a = (double)dY / dX;
        b = (double)from.getY() - (a*from.getX());
        tilted = false;

    // vertical line
    } else {
        a = (double)dX / dY;
        b = (double)from.getX() - (a*from.getY());
        tilted = true;
    }
}
```

Die Geraden befinden sich nun in einer geeigneten Form und der Schnittpunkt kann berechnet werden. Sind beide beteiligen Geraden gekippt oder beide nicht gekippt, kann mit der nachfolgenden Formel der Schnittpunkt berechnet werden. Falls die beiden Geraden gekippt waren, müssen x und y vertauscht werden.

$$x = \frac{b_2 - b_1}{a_1 - a_2} \qquad y = a_1 * x + b_1$$

Der Ansatz zum Berechnen des Schnittpunktes bei einer gekippten und einer nicht gekippten Geraden ist ähnlich. Durch das Einsetzen von $y = a_1 * a_1 + b_1$ in $x = a_2 * y + b_2$ konnten die nachfolgenden Formeln hergeleitet werden. Falls die erste Gerade gekippt war, müssen auch hier x und y vertauscht werden.

$$y = \frac{a_1 * b_2 + b_1}{1 - a_1 * a_2} \qquad x = \frac{y - b_1}{a_1}$$

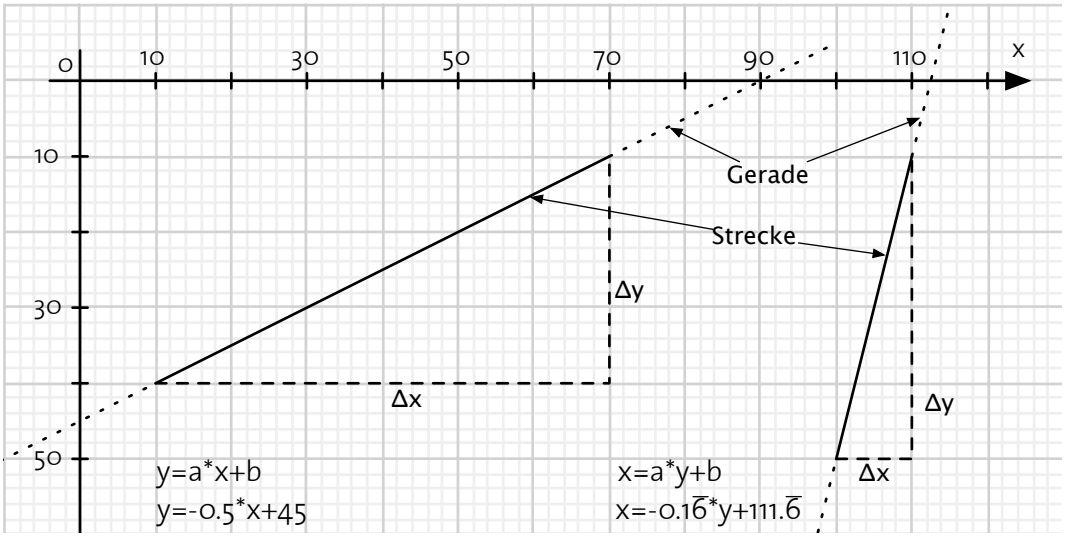


Abbildung 4.27: Geradengleichung gekippten und nicht gekippten Linien.

Durch diese Formeln konnte folgende Methode geschrieben werden, die jeden möglichen Fall abdeckt und jederzeit einen Schnittpunkt zurückgeben kann. Für den Spezialfall, dass die beiden Geraden parallel verlaufen, wird `null` zurückgegeben, in jedem anderen Fall die Koordinaten des Schnittpunkts. Ob diese auf der Strecke oder ausserhalb liegen wird dabei nicht beachtet.

```
public Point intersection(Edge other) {
    if (this.tilted == other.tilted) {
        // check if parallel
        if (this.a == other.a) {
            return null;
        }

        // x coordinate of point of intersection (if not tilted)
        int x = (int)Math.round((other.b - this.b) / (this.a - other.a));

        if (! this.tilted) {
            return new Point(x, (int)(this.a*x + this.b));
        } else {
            return new Point((int)(this.a*x + this.b), x);
        }

    } else { // non-identical tilting
        // y coordinate of point of intersection (if not tilted)
        int y = (int)Math.round((this.a * other.b + this.b)
                                /(1-this.a*other.a));
        if (! this.tilted) {
            return new Point((int)((y-this.b)/this.a), y);
        } else {
            return new Point(y, (int)((y-this.b)/this.a));
        }
    }
}
```

Obige Methode vergleicht Geraden und findet immer einen Schnittpunkt, ausser wenn die beiden Geraden absolut parallel liegen. Für das Spiel müssen jedoch Strecken, keine Geraden, verglichen werden. Hat man den Schnittpunkt zweier Geraden, muss daher noch überprüft werden, ob der Schnittpunkt auf der Strecke liegt. Folgende Methode übernimmt diese Aufgabe.

```
public boolean intersects(Edge other) {
    Point intersection = intersection(other);

    if (intersection == null) {
        return false;
    } else {
        if (!tilted) {
            int xIntersec = intersection.getX();
            return ((from.getX()-xIntersec) * (to.getX()-xIntersec) < 0) &&
                   ((other.from.getX()-xIntersec) * (other.to.getX()-
                       xIntersec) < 0);
        } else {
            int yIntersec = intersection.getY();
            return ((from.getY()-yIntersec) * (to.getY()-yIntersec) < 0) &&
                   ((other.from.getY()-yIntersec) * (other.to.getY()-
                       yIntersec) < 0);
        }
    }
}
```

Obwohl das oben beschriebene Vorgehen funktioniert, wurde die oben gezeigte Methode `intersects(Edge)` später durch folgenden Code ersetzt:

```
public boolean intersects(Edge other) {
    return Line2D.linesIntersect(this.from.getX(), this.from.getY(),
        this.to.getX(), this.to.getY(), other.from.getX(),
        other.from.getY(), other.to.getX(), other.to.getY());
}
```

Die Methode `Lines2D.linesIntersect(int, int, int, int)` ist im Paket `java.awt.geom` verborgen, weshalb diese zuerst nicht gefunden wurde. Die selbstgeschriebene Methode zum Berechnen des Schnittpunkts ist jedoch trotzdem nötig, da die in Java integrierte Klasse den Schnittpunkt nicht zurückgeben kann. Der exakte Schnittpunkt wird bei einigen Regeln in der Spiellogik gebraucht.

The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with today's machines and what we have so far been able to finish.

Donald Ervin Knuth

5 Fazit

Dieses Kapitel beginnt mit einer kurzen Zusammenfassung, die die wichtigsten Punkte nochmals wiederholt. In 5.1 werden die Resultate aufgezeigt, bevor im nächsten Abschnitt auf mögliche weitere Projekte in diesem Umfeld eingegangen wird. Den Abschluss bildet auf Seite 72 ein persönliches Fazit des Autors.

Ziel der Arbeit war das Erweitern des Spiels WH40k mittels RFID-Technologie, wobei die Antenne mittels eines Roboters unter dem Spielfeld bewegt wird. Neben dem Bau des Spielfelds sollte auch ein Computerprogramm entwickelt werden, das auf dem Bildschirm das physikalische Spielfeld nachbildet und Informationen zu den einzelnen Figuren ausgeben kann.

Nebst der Anzeige der Informationen müsste das Programm anhand der Regeln feststellen können, welche Spielzüge gültig sind und den Spielern das Messen von Abständen, Zählen von Einheiten und Nachschlagen von Attributen abnehmen.

5.1. Resultate

Die Idee mit der bewegten Antenne hat sich als äusserst interessant herausgestellt. Die gemessenen Abweichungen von nur ca. $2mm$ beim Abtasten des Spielfelds waren wesentlich besser als erwartet. Es ist durchaus möglich, dass mit dem Einsatz besserer Hardware die Abweichung noch weiter reduziert werden kann. Die LEGO-Elemente haben sich bei der Konstruktion gut bewährt, aber es hat sich auch gezeigt, wo die Grenzen liegen. Bei dieser Auflösung dauert das Einlesen des Spielfelds viel zu lange. Ohne massive Umbauten an der Hardware ist eine schnellere Abtastung kaum machbar. Auch das manuelle Ausmessen der Position hat sich als unerwartet schwierig herausgestellt, da die ganze Konstruktion durch das Anheben des Spielfelds an Festigkeit verloren hat.

Die Steuerung der LEGO-Hardware vom PC aus hat, bis auf kleinere Probleme mit der erst kürzlich hinzugefügten Unterstützung für OS X, gut funktioniert. Die von LEGO veröffentlichte Dokumentation, wie zum Beispiel „NXT Communication Protocol“, hat sich dabei als sehr praktisch erwiesen. Durch die kurze Verzögerung eignet sich USB sehr gut für solche Projekte, bei denen das NXT-Modul ferngesteuert wird.

Für das Ansprechen des RFID-Lesegeräts konnten aus älteren Projekten die benötigten Informationen extrahiert werden. Tückisch war der fehlerhafte Treiber des Herstellers des Adapters von USB auf seriell und die Tatsache, dass das RFID-Lesegerät mit zu schnell aufeinanderfolgenden Anfragen nicht klarkommt. Nach dem Lösen dieser Probleme hat das Lesegerät sehr zuverlässig bis zu acht Tags gleichzeitig erkannt. Die Integration über JNI funktionierte für die USB-Kommunikation mit dem NXT-Modul und auch für die Verbindung mit dem Lesegerät über RS-232 gut.

Die Verarbeitung der Daten von der Hardware hat dank den eingeführten Referenz-Tags auf dem Spielfeld problemlos funktioniert. Der entwickelte Algorithmus zum Erkennen der Ausrichtung von Objekten mit mehreren Tags hat sich, nach unerwarteten Problemen mit dem Mitteln der Winkelwerte, als sehr robust erwiesen.

Die XML-Dateien haben sich gut bewährt zum Speichern der Spielobjekte. Speziell die einfache Möglichkeit zum Ändern hat sich bezahlt gemacht. Das Laden der Spielobjekte aus den XML-Dateien konnte wie geplant realisiert werden, enthält bisher jedoch nur wenige verschiedene Modelle.

Für die grafische Benutzerschnittstelle wurden einzelne `JComponent`-Objekte für die verschiedenen Modelle instanziert und auf dem Bildschirm gezeichnet, was das Ansprechen der einzelnen Modelle sehr vereinfacht hat. Die Grafik wurde vorbereitet, um ein Bild des Spielfelds hinter den Modellen anzuzeigen. Dies konnte jedoch aus zeitlichen Gründen nicht mehr realisiert werden.

Für Hilfsfunktionen der Spiellogik, wie beispielsweise das Erkennen aller Objekte auf einer Linie, konnten funktionierende Algorithmen gefunden und auch implementiert werden. Als grösste Hürde dieses Projekts hat sich das überaus komplexe Regelwerk von WH40k erwiesen. So konnte durch das Aufteilen der Spiellogik in sechs verschiedene Module die Übersicht verbessert werden, aber selbst bei diesem kleinen Teil der implementierten Regeln wurden die Klassen eher unübersichtlich. Die Schwierigkeit war das Abbilden der als endliche Automaten definierten Abläufe in Java. Durch die vielen benötigten Interaktionen mit den Spielern ist der Programmfluss nicht mehr offensichtlich.

Das für diese Art von Spiel sehr wichtige Würfeln musste vollständig simuliert werden. Es werden einfach zu viele Würfel gleichzeitig benötigt. Wie das gelöst werden kann, müssen zukünftige Projekte zeigen. Auch müssen die Spieler stark mit dem Computer interagieren, was zu sehr den sozialen Aspekt des Spiels verdrängt. Der Computer wird zum Mittelpunkt anstatt zu einem Hilfsmittel. Selbst der Spielablauf lässt sich nicht nur über das Abtasten des Spielfelds erfassen. Das Programm benötigt weitere Informationen über den Spielverlauf, um überhaupt ein nutzbares Bild des aktuellen Zustands zu erhalten.

Die Tabelle (Tabelle 3.1, Seite 14) enthält die Anforderungen an das System und wird hier der Übersichtlichkeit wegen wiederholt.

Tabelle 5.1.: Anforderungen (Priorität: A(höchste) bis C, X ist eine Voraussetzung)

Nummer	Titel	Beschreibung	Prio
1	Hardware	LEGO Mindstorms und Feig RFID-Lesegerät sollen genutzt werden	X √
2.a	Spielfeld	Die RFID-Antenne soll unter dem Spielfeld bewegt werden	X √
2.b		Die Abweichung beim Abtasten des Spielfelds sollten höchstens $1cm$ betragen.	A √
2.c		Kleine Modelle werden mit einem RFID-Tag markiert, grössere mit mindestens zwei	X √
3	Anzahl Spieler	Es gibt mindestens zwei Spieler, keinen Computergegner	A √
4	Spielphasen	Die Phasen *Bewegen*, *Schiessen* und *Nahkampf* sollten implementiert sein	X √
4.1.a	Phase: Bewegen	Funktioniert grundsätzlich, testet auf Kohärenz der Einheit sowie zurückgelegtem Weg	A √
4.1.b		Gefährliches und schwieriges Terrain wird beachtet	B √
4.1.c		Überlappung mit anderen Modellen wird erkannt	B [1]
4.1.d		Berechnen der Weglänge um Hindernisse herum anstatt nur Luftlinie	C ∅
4.2.a	Phase: Schiessen	Funktioniert grundsätzlich, testet auf Distanz	A √
4.2.b		Testen auf Sichtkontakt (keine Objekte im Weg)	B √
4.2.c		Testen der Höhe der betroffenen Objekte	C ∅
4.2.d		Beachten der Deckung	C ∅
4.3.a	Phase: Nahkampf	Funktioniert grundsätzlich	A √
4.3.b		Beachten der Deckung	C ∅
5.a	GUI	Darstellen des Spielfelds und der Einheiten	A √
5.b		Anzeigen des Zustands aller Objekte	B √
5.c		Hinterlegen des Spielfelds mit einem Bild des echten Spielfelds	B [2]
6	Konfiguration	Konfigurationsänderungen sollten möglich sein ohne Neukompilieren des Programms	A √
7.a	Zukunft	Vollständige Dokumentation des Quelltexts	A √
7.b		Vorbereiten des Programms für spätere Erweiterungen	B [3]

[1] Funktioniert nur mit einfachen Modellen. Die Überlappung mit Fahrzeugen ist nicht implementiert.
[2] Das Programm ist darauf vorbereitet, aber das eigentliche Einblenden des Bilds ist nicht implementiert.
[3] In der Planung ist viel Wert auf Erweiterbarkeit gelegt worden. Diese Eigenschaft lässt sich jedoch nicht testen.

Mit $\surd$ markierte Anforderungen konnten vollständig implementiert werden, während das Symbol $\emptyset$ nicht implementierte Anforderungen kennzeichnet. Die Tabelle zeigt, dass alle Voraussetzungen und die Anforderungen mit Priorität A erfüllt wurden, von denen mit Priorität B und C nur ein Teil. Einige dieser Anforderungen konnten zumindest teilweise integriert werden.

Die Überlappung von Einheiten (Anforderung 4.1.c) konnte nur bei den einfachen Modellen mit rundem Fuss implementiert werden. Für das Erkennen einer Überlappung mit Fahrzeugen könnten Java-Klassen aus `java.awt.geom` genutzt werden. Diese bieten einen grossen Teil der benötigten Funktionen.

Wie bereits erwähnt, wurde das Spielfeld auf ein Bild im Hintergrund (Anforderung 5.c) vorbereitet. Java-Klassen zum Ansprechen einer USB-Videokamera sind bei der Firma Morena erhältlich. Eine Education-Lizenz liegt den elektronischen Dokumenten dieser Arbeit bei.

Ob die Vorbereitungen für zukünftige Erweiterungen (Anforderung 7.b) umfassend genug sind, kann erst in zukünftigen Projekten endgültig festgestellt werden.

Abschliessend lässt sich sagen, dass das eigentliche Spielfeld die Erwartungen übertroffen hat. Das geschriebene Programm erfüllt die Anforderungen, benötigt jedoch aus verschiedenen Gründen zu viel Aufmerksamkeit von den Spielern, was den Spielverlauf negativ beeinflusst. Obwohl noch viel Zeit investiert werden muss, entstand mit dieser Arbeit eine gute Plattform für weitere Projekte. Einige Vorschläge für mögliche Erweiterungen folgen im nächsten Abschnitt.

5.2. Weitere Arbeiten

Während der Arbeit haben sich verschiedenste mögliche Erweiterungen gezeigt, die nicht mehr integriert werden konnten. Die folgenden Vorschläge wurden unterteilt in verschiedene Kategorien. Die erste Kategorie beschäftigt sich mit Erweiterungen an der Benutzerschnittstelle. Das Ziel ist, den Fokus vom Computer zurück zum eigentlichen Spiel zu bringen. Die zweite Kategorie zeigt Erweiterungen an der Technik von LEGO und den RFID-Tags auf, die den Einsatz dieses Programms vereinfachen würden. Zuletzt folgen noch einige Ideen, die sich nicht in eine der ersten beiden Kategorien einteilen lassen.

5.2.1. Erweitern der Benutzerschnittstelle

- Der Würfelsimulator muss abgelöst werden. Am Besten wäre ein Erkennen physikalischer Würfel. Vielleicht liesse sich über Bilderkennung die Anzahl Punkte auf dem Würfel erkennen. Durch den hohen Kontrast und die nur wenigen möglichen Muster ist dies sehr erfolgsversprechend.
- Ein aktuelles Bild des Spielfelds kann als Hintergrundbild in das virtuelle Spielfeld integriert werden. Das Programm ist bereits darauf vorbereitet. Aufgrund der Komplexität beim Skalieren, Rotieren und Stauchen der Bilder von nicht exakt über dem Spielfeld angebrachten Kameras wurde dies nicht vollständig integriert.

- Die benötigte Zeit zum Scannen des Spielfelds muss reduziert und das Spielfeld vergrössert werden. Dazu liessen sich mehrere dieser bereits bestehenden Spielfelder kombinieren, oder man könnte auf den Arm des Roboters mehrere Antennen montieren. Vermutlich erhielte man die besten Werte, wenn man LEGO durch eine eigens für dieses Projekt entwickelte Hardware ersetzen würde.
- Könnte die Zeit stark reduziert werden, kann auf weitere Benutzerinteraktion verzichtet werden. Anstatt Schütze und Ziel auf dem Bildschirm anzuklicken, könnte dies mit Markern gelöst werden. Diese Marker würden mit RFID-Tags versehen und lassen sich daher auf dem Spielfeld erkennen. Dies würde den Fokus wieder mehr auf das physikalische Spielfeld lenken. Auch das Wechseln der Spielphase könnte gut über einen speziellen Bereich neben dem Feld gelöst werden. Man müsste einen Marker auf das Feld für die aktuelle Spielphase stellen. Wird eine Phase abgeschlossen, wird die Figur weiterbewegt.
- Dieser Marker könnte auch mit einfachen LEDs ausgestattet werden, die anzeigen, ob auf die gewählte Einheit geschossen werden kann. Noch besser wäre eine LCD-Anzeige, die anzeigt, wie viele Schüsse möglich sind. Diese Marker könnten über Induktion in der Aufbewahrungsbox jeweils aufgeladen werden.
- Möglich wäre auch die Integration eines kleinen Bildschirms in das Spielfeld. Auf diesem könnten jeweils aktuell benötigte Attribute von Einheiten, aber auch Hinweise bei ungültigen Spielzügen angezeigt werden.

5.2.2. Techik

- Aktuell werden für das NXT-Modul von LEGO sehr viele Batterien benötigt. Wird weiterhin LEGO eingesetzt, muss ein Netzadapter konstruiert werden. Dadurch löst sich auch die Problematik mit den Motoren, die je nach Ladestand der Batterien mehr oder weniger kräftig sind.
- Anstatt die Attribute zu Spielobjekten in einer XML-Datei zu speichern, könnten diese direkt im RFID-Tag abgelegt sein. Vielleicht $100 - 200$ Bytes würden genügen für sämtliche Attribute. Es wäre auch möglich, nur den Typ der Einheit abzuspeichern, was weniger flexibel ist, dafür keinen Missbrauch durch die Spieler zulassen würde. Eine spezielle Hardware-Komponente könnte zum Speichern der Daten auf den Tag erstellt werden. So könnten die Tags sehr günstig in grösseren Mengen gekauft werden, und die Spieler legen selbst fest, welche Eigenschaften diesem Tag zugewiesen werden. Nur über diesen Weg ist es auch möglich, dass verschiedene Spieler ihre eigenen Figuren an ein Spiel mitbringen und diese sofort genutzt werden können. Der jetzige Ansatz mit einer XML-Datei skaliert sehr schlecht.
- Bisher wurde nicht beachtet, dass Einheiten auf höheren Ebenen (Gebäuden) stehen können. Falls diese Einheiten vom RFID-Lesegerät unter dem Spielfeld überhaupt noch erkannt werden, könnte man über die Anzahl erfolgreicher Lesevorgänge die Höhe abschätzen. Eine Einheit, die auf dem Spielfeld steht, wird ungefähr 30 Mal gelesen, ist die Einheit weiter oben reduziert sich diese Zahl.

5.2.3. Diverses

- Die Daten des Programms ArmyBuilder könnten integriert werden. Damit liesse sich direkt auf dem Spielfeld feststellen, ob die aktuell aufgestellte Armee sämtliche Anforderungen erfüllt.
- Die vielen weiteren möglichen Modelle und auch die restlichen Regeln sollten noch implementiert werden. Speziell bei den Fahrzeugen besteht noch viel Nachholbedarf.
- Ein XML-Schema für die benötigten Konfigurationsdateien würde helfen, Fehler in diesen Dateien schnell zu erkennen. Der grösste Teil dieser Dateien liesse sich jedoch ablösen durch das Speichern der Daten auf den Tags.

Die Idee eines elektronischen Spielfelds hat viel Potenzial und wurde von den interviewten Spielern auch durchaus begrüsst. Die wichtigsten Punkte sind klar das Beschleunigen des Einlesens, das Vervollständigen der Regeln und das Finden einer stabileren, einfachen Konstruktion zum Bewegen der Antenne unter dem Spielfeld.

5.3. Persönliches Fazit

Die Suche nach einem geeigneten Thema für die Masterarbeit war wesentlich schwieriger als erwartet. Als Steve Hinske mir dieses Projekt vorgeschlagen hat, war ich zuerst nicht wirklich begeistert. Ich habe nie solche Spiele wie WH40k gespielt und konnte mich damit auch nicht identifizieren. Was mich jedoch reizte, war das RFID-Spielfeld und auch die Vielfältigkeit dieses Projekts. So kam es, dass ich trotz anfänglicher Zweifel zusagte.

Sehr speziell war für mich der Einstieg über die Konstruktion des Spielfelds mit LEGO. Als Kind hatte ich oft mit LEGO gespielt und hätte niemals erwartet, dass ich für die Masterarbeit die alten LEGO-Steine wieder hervorholen würde. Beim Suchen einzelner Teile kamen viele alte Erinnerungen wieder hoch.

Auch das Ansprechen der Hardware über den Computer war sehr interessant. Ich hatte schon länger nichts mehr mit C zu tun, und es waren meine ersten Erfahrungen mit dem Steuern von Hardware, was mich total faszinierte. Es ist unglaublich, was heute mit LEGO und einem PC alles möglich ist.

Ein schwieriger Teil war für mich das Kennenlernen von WH40k. Obwohl ich inzwischen verstehen kann, was die Spieler daran fasziniert, wird das bestimmt nie ein Hobby für mich werden. Schon vor Jahren habe ich mich von Computerspielen getrennt, und wenn ich doch einmal wieder eines hervorhole, dann nur für kurze Zeit. Ein Spiel wie WH40k ist mir viel zu zeitintensiv, und ich kann mich damit auch nicht identifizieren. Dies führte dann auch zu einer Phase, in der ich mit mangelnder Motivation zu kämpfen hatte.

Viel besser lief es mir bei der eigentlichen Programmierung. Ich programmiere ganz gerne und konnte mit Java auch bereits einige Erfahrungen sammeln. Das Ansprechen der Hardware, das Programmierern der grafischen Oberfläche, aber auch das Festlegen der Architektur war zu einem grossen Teil eine sehr interessante Arbeit.

Besonders fasziniert hat mich beim Programmieren das Suchen nach Algorithmen. Sei dies für die Erkennung von Schnittpunkten auf Linien oder das Erkennen der Position und Ausrichtung von Objekten mit mehreren Tags. Die Tücken lagen jeweils im Detail, aber irgendwann hat es immer funktioniert.

Das Einarbeiten der Spielregeln in das vorhandene Grundgerüst für das Programm war der langwierigste und mühsamste Teil für mich. Ein kleines Teilchen nach dem anderen und weit und breit kein Ende in Sicht. Auch war diese Arbeit überhaupt nicht interessant, sondern eine reine Fleissarbeit. Ich war froh, dass ich diese Arbeit mit dem Schreiben der Dokumentation, aber auch mit einem Treffen mit langjährigen Spielern von WH40k unterbrechen konnte.

Dieses Treffen mit den Spielern war sehr interessant. So konnte ich erst ab diesem Abend, wo ich bei einem Spiel zuschauen durfte, die Motivation für das Spiel und auch dessen Charakter richtig erkennen. Das Regelbuch versucht teilweise romanartig das Spielgeschehen zu vermitteln, was aber, zumindest bei mir, überhaupt nicht funktioniert hat. Es war sogar so, dass diese Schreibweise im Regelwerk äusserst lästig war, wenn ich die reinen Fakten herauslesen wollte. An dieser Stelle einen herzlichen Dank an die beiden Spieler, die mich offen empfangen und gerne alle meine Fragen beantwortet haben. Ich bereue nur, dass ich mich nicht früher mit ihnen getroffen hatte.

Am Anfang scheinen einem die sechs Monate sehr lang zu sein, doch bald schon sind fünf Monate vorbei, und man muss sich mit der Dokumentation beschäftigen. Es war durchaus interessant, anhand der Notizen die letzten Monate nochmals Revue passieren zu lassen und während des Schreibens der Texte noch einige Grafiken zu erstellen. An dieser Stelle möchte ich mich bei Rolf Schär und meiner Schwester Sibylle für das Korrekturlesen bedanken.

Ein besonderer Dank geht an Steve Hinske. Von ihm stammt nicht nur die Idee für das ganze Projekt, sondern er hat mich auch tatkräftig mit Ideen und Antworten auf alle meine vielen Fragen unterstützt.

Alles in allem ein interessantes, abwechslungsreiches Projekt, bei dem ich viel über das Thema, aber auch über mich selbst lernen konnte. Ein würdiger Abschluss meiner Zeit an der ETH Zürich.

A

Programm mit Simulator starten

Zum Testen des Programms eignet sich der Simulator am besten, da so am wenigsten Fehler auftreten können. Für das Ansprechen der Hardware werden einige JNI-Bibliotheken benötigt, die erst kompiliert werden müssen. Wie dies funktioniert, wird in Anhang B beschrieben.

A.1. Anpassen der Konfiguration

Die Konfigurationsdatei `configuration.xml` muss für einen Simulator-Betrieb konfiguriert sein. Um dies zu erreichen, sind die beiden Beans mit den IDs `GameField` und `CoordTranslator` entsprechend zu konfigurieren. Andere Definitionen dieser Bean-IDs müssen gelöscht oder als Kommentar markiert werden.

Die ausgelieferte Konfigurationsdatei sollte bereits für den Betrieb mit dem Simulator vorbereitet sein. Die Angaben unten dienen als Referenz.

Zuerst wird der eigentliche Simulator initialisiert:

```
<!-- Initialization for RFID-Simulator ->
<bean id="GameField" factory-method="startSimulator"
 class="ch.ethz.lthomas.ma.rfidsimulator.RFIDSimulatorApp">
    <constructor-arg><ref bean="CoordTranslator"/></constructor-arg>
</bean>
```

Der Koordinatentransformator muss die im Simulator verwendeten Referenztags kennen. Die korrekte Konfiguration dafür sieht wie folgt aus:

```
<!– Initialization for Coordinate–Translator. CONFIGURED FOR SIMULATOR! –>
<bean id="CoordTranslator" class="ch.ethz.lthomas.ma.core.CoordTranslator">
    <!– First reference point with tag–id and real x/y coordinates –>
    <constructor–arg index="0" type="java.lang.String" value="ref1"/>
    <constructor–arg index="1" type="int" value="0"/>
    <constructor–arg index="2" type="int" value="0"/>

    <!– Second reference point with tag–id and real x/y coordinates –>
    <constructor–arg index="3" type="java.lang.String"  value="ref2"/>
    <constructor–arg index="4" type="int" value="1000"/>
    <constructor–arg index="5" type="int" value="1000"/>

    <constructor–arg><ref bean="TagInterpreter"/></constructor–arg>
</bean>
```

A.2. Bedienung des Simulators

Der Simulator wird automatisch gestartet, wenn das WH40k-Programm entsprechend konfiguriert ist.

Ein Doppelklick auf das Spielfeld ermöglicht das Hinzufügen von neuen Tags auf dem Spielfeld. Die ID aller Tags müssen dabei zu einem konfigurierten Modell passen. Die Konfiguration wird im Kapitel C.5 genauer beschrieben. Die Referenz-Tags müssen nicht erfasst werden, da diese vom Simulator automatisch konfiguriert werden.

Durch Klicken und Ziehen mit der Maus können vorhandene Tags verschoben werden. Das Spiel erhält sofort die neuen Koordinaten. Wird ein Tag links oder oben aus dem Spielfeld herausgezogen, entspricht dies dem Entfernen eines Modells vom Spielfeld. Solche Tags erscheinen in der Liste auf der rechten Seite und können von dort wieder zurückgebracht oder endgültig gelöscht werden.

Durch einmaliges Klicken auf einen Tag kann dieser markiert werden. Wird gleichzeitig die Control-Taste gedrückt, können auch mehrere Tags markiert werden. Sämtliche markierte Tags werden gleichzeitig verschoben, wenn einer der Tags verschoben wird.

Der Simulator und das WH40k-GUI verwenden je eine andere Skalierung des Spielfelds. Die Grafiken A.1 und A.2 auf der nächsten Seite zeigen ein typisches Spielszenario. Der Simulator weiss nichts von den speziellen Gebieten auf dem Spielfeld. Daher wird der rote Bereich, der in `areas.xml` konfiguriert ist, auch nicht angezeigt.

Abbildung A.1.: RFID-Spielfeld Simulator zu Grafik A.2

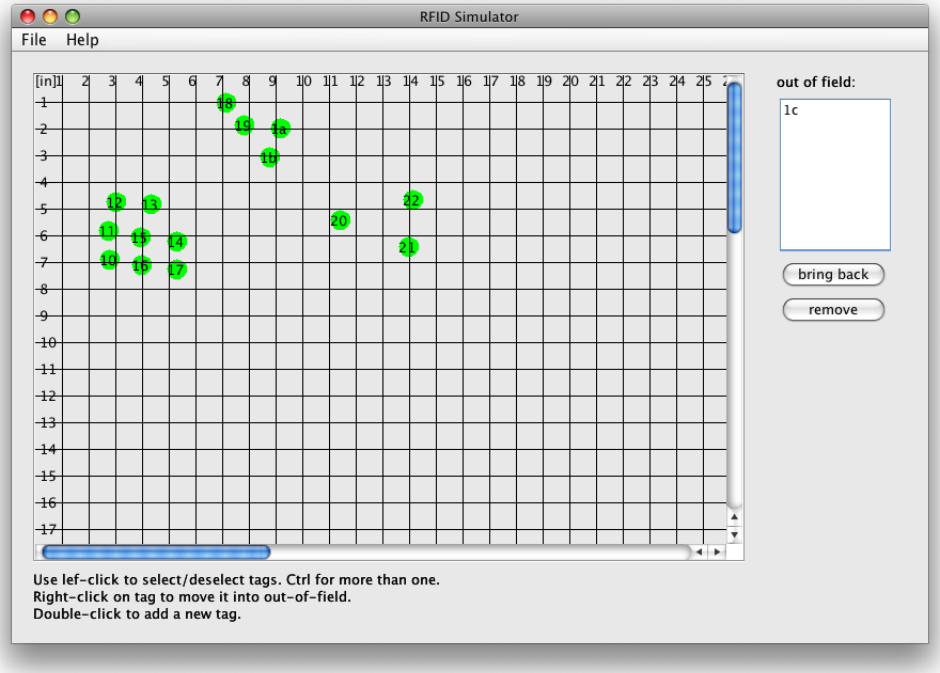
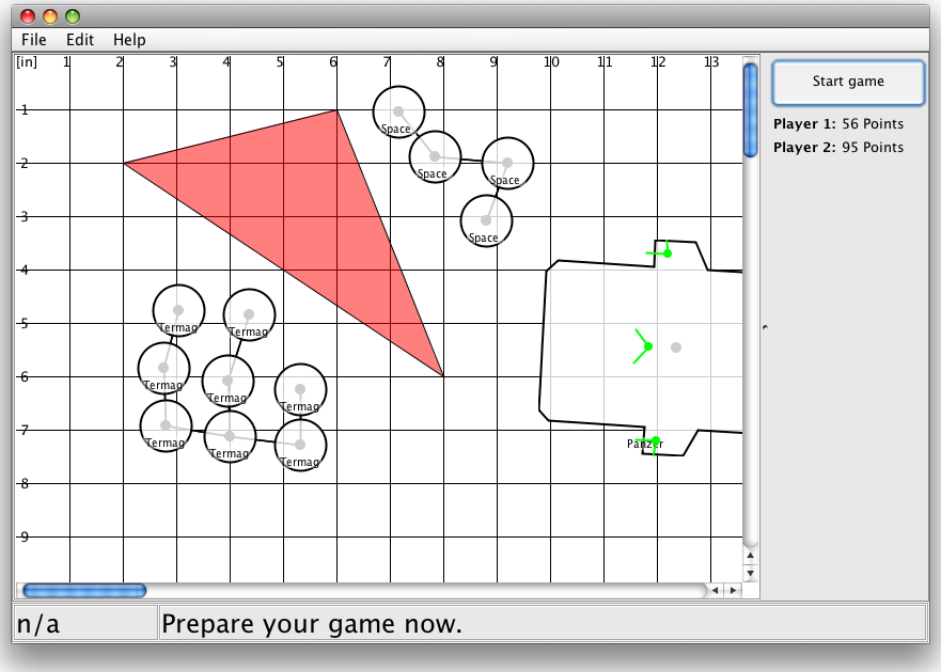


Abbildung A.2.: GUI zu Simulator A.1



B

Programm mit RFID-Spielfeld starten

Man sollte das Programm immer zuerst mit dem Simulator testen, wie in Anhang A beschrieben. Beim Betrieb mit der Hardware gibt es wesentlich mehr mögliche Fehlerquellen.

B.1. RFID-Spielfeld vorbereiten

Die komplette Schritt-für-Schritt Aufbauanleitung für das LEGO-Modell befindet sich in der Datei `Modell (mit Feld).ldr`. Es handelt sich um eine Datei im standardisierten `LDRAW`-Format. Abbildung B.1 zeigt nur wenige der benötigten Schritte für den Aufbau der wichtigsten Komponente.

Wurde das Modell nach dieser Anleitung zusammengesetzt, müssen die Motoren und Sensoren wie folgt mit der NXT-Steuereinheit verbunden werden:

- Port A der NXT-Einheit wird mit dem unteren Motor, der die Kette antreibt, verbunden.
- Port B der NXT-Einheit wird mit dem oberen Motor, der den Arm bewegt, verbunden.
- Port 1 der NXT-Einheit wird mit dem Tastsensor unter der NXT-Einheit verbunden.
- Port 2 der NXT-Einheit wird mit dem Reflektionssensor verbunden, der die weisse Stelle auf dem Arm erkennt.
- Die NXT-Einheit wird über USB mit dem Computer verbunden.

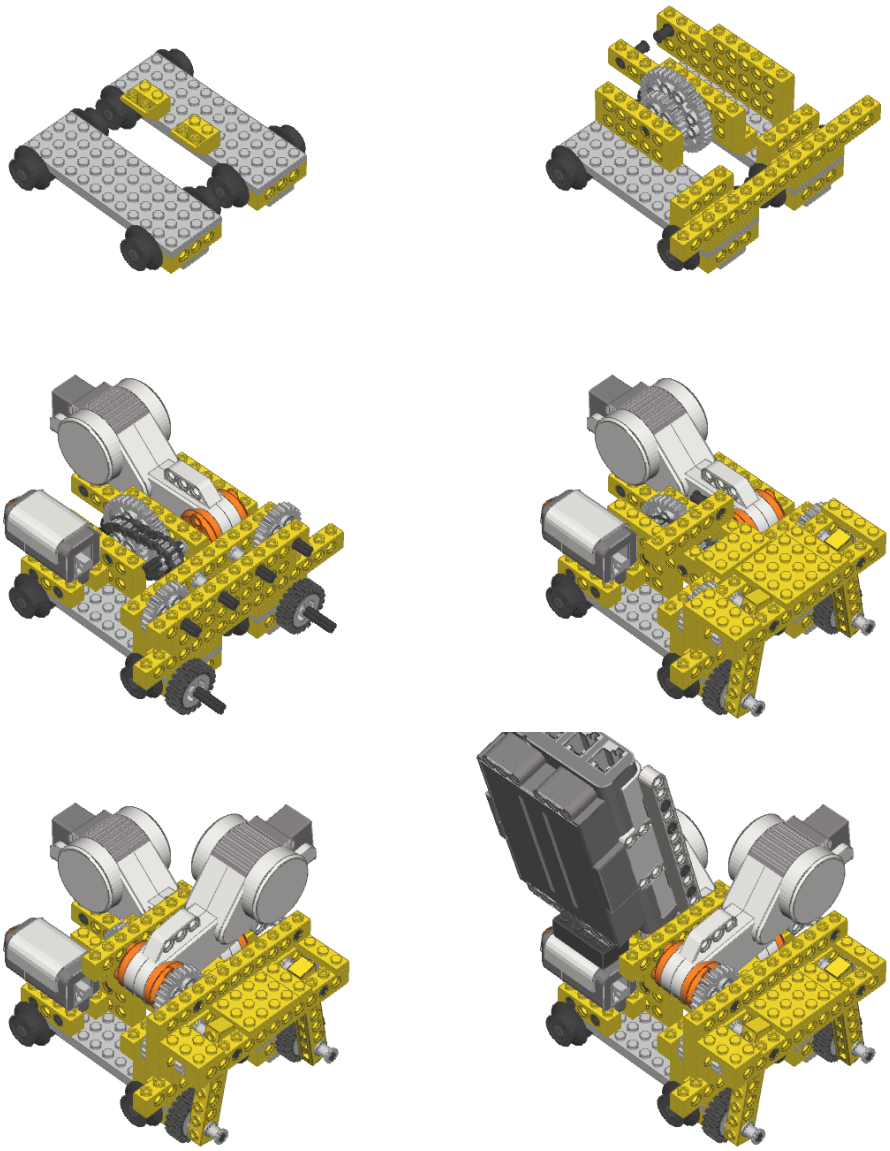


Abbildung B.1: Schritte zum Aufbau des LEGO-Modells

B.2. Benötigte Bibliotheken kompilieren

Da in Java die Unterstützung für direkte Hardwarezugriffe fehlt, mussten die Zugriffe auf die LEGO NXT-Steuereinheit und das RFID-Lesegerät in C implementiert werden. Für OS X sind sämtliche benötigten Bibliotheken bereits vorbereitet. Unter Linux oder Windows müssen diese zuerst kompiliert werden. Die Hardwarezugriffe wurden ausschliesslich auf Mac OS X Leopard getestet. Andere UNIX-Systeme sollten funktionieren. Für Windows sind möglicherweise Anpassungen am Quelltext nötig.

Der nächste Abschnitt beschreibt das Vorgehen zum Erstellen der beiden Bibliotheken auf UNIX-Systemen wie beispielsweise Linux.

Feig

Das vorhandene Modul `FeigJNI-native` unterstützt ausschliesslich die RFID-Lesegeräte `MR101` der Firma Feig Electronic. Der Konsolenbefehl `make` erstellt die Bibliothek für das eigene Betriebssystem und speichert diese in einem Unterverzeichnis von `dist`.

Das NetBeans-Projekt `FeigJNI` enthält die nötigen Hilfsklassen, um Java den Zugriff auf das RFID-Lesegerät zu ermöglichen. Die oben erstellte Bibliothek muss in das Verzeichnis `libs` kopiert und das Projekt neu kompiliert werden. Die daraus entstehende Datei `FeigJNI.jar` muss nun wiederum in das Verzeichnis `libs` des NetBeans-Projekts WH40k kopiert werden.

NXT

Auch dieses Modul besteht aus zwei Komponenten. Die erste Datei `pccomm.jar` stammt vom LEJOS-Projekt [3] und kann direkt übernommen werden. Für die Kommunikation über USB wird eine installierte `libusb` [13] benötigt. Die Version `0.1.12` lässt sich problemlos mit `./configure && make && sudo make install` installieren. Zusätzlich muss der Treiber von LEGO [2] installiert werden.

Der zweite Teil befindet sich im NetBeans-Projekt NXT. Es handelt sich um die entsprechenden Steuerbefehle für die Low-Level-Kommunikationsroutinen aus `pccomm.jar`. Nach dem Kompilieren erhält man die die Datei `NXT.jar`, die in das Verzeichnis `libs` des NetBeans-Projekts WH40k kopiert werden muss.

B.3. Anpassen der Konfiguration

Erst müssen die IDs und die Positionen der beiden Referenz-Tags bekannt sein. Diese sollten möglichst an diagonal gegenüberliegenden Ecken auf dem Feld angebracht werden. Am einfachsten ist dies über das Kommandozeilenprogramm `feigread-standalone` möglich. Beim Starten muss nur die serielle Schnittstelle angegeben werden. In diesem Beispiel ist dies `/dev/tty.PL2303-004052FD`.

```
$ ./feigread-standalone /dev/tty.PL2303-004052FD
Opening port '/dev/tty.PL2303-004052FD'...
Port open.
Port configured.
Reset completed. Continue in 2 seconds.
Init completed..
>  [137,157,197,160]
>  [137,157,197,159]
> 0900000081B19CE6# [137,160,231,231]
> 0900000081B19CE6# [137,161,231,195]
> 0900000081B19CE6# [137,162,231,190]
> 0900000081B19CE6# [137,163,231,200]
> 0900000081B19CE6# [137,164,231,190]
```

Bei den ersten beiden Lesevorgängen wurde kein Tag erkannt, bei den fünf folgenden befand sich jeweils der Tag mit der ID `0900000081B19CE6` in Reichweite. Das `#` dient nur als Trennzeichen bei mehreren Tags, und die Zahlen beinhalten Informationen über die benötigte Zeit.

Sind nun die ID und auch die Position beider Referenz-Tags bekannt, kann der Koordinatentransformator wie folgt konfiguriert werden:

```
<!--
  Initialization for Coordinate-Translator. CONFIGURED FOR HARDWARE
  Use 1/100in for x/y coordinates in attribute value.
  -->
<bean id="CoordTranslator" class="ch.ethz.lthomas.ma.core.CoordTranslator">
    <!-- First reference point with tag-id and real x/y coordinates -->
    <constructor-arg index="0" type="java.lang.String"
     value="0900000081B19A22"/>
    <constructor-arg index="1" type="int" value="0"/>
    <constructor-arg index="2" type="int" value="0"/>

    <!-- Second reference point with tag-id and real x/y coordinates -->
    <constructor-arg index="3" type="java.lang.String"
     value="0900000081B1A8A1"/>
    <constructor-arg index="4" type="int" value="790"/>
    <constructor-arg index="5" type="int" value="790"/>

    <constructor-arg><ref bean="TagInterpreter"/></constructor-arg>
</bean>
```

Die hier vorhandenen IDs müssen jeweils durch die IDs der eigenen Tags ersetzt und die entsprechende reale Position in $\frac{1}{100}$ Zoll angegeben werden.

Möglicherweise vom Simulatorbetrieb vorhandene Bean-Konfigurationen mit der selben ID müssen unbedingt auskommentiert werden. Dies gilt beim Koordinatentransformator ebenso wie beim Spielfeld.

Die Konfiguration des eigentlichen Spielfelds benötigt nur den Pfad zur seriellen Schnittstelle. Das über USB verbundene NXT-Gerät wird automatisch erkannt.

```
<!--
  Initialization for RFID-Gamefield
  -->
 <bean id="GameField"
  class="ch.ethz.lthomas.ma.hardware.SmartRFIDField">
    <constructor-arg>
        <bean class="ch.ethz.lthomas.ma.nxt.NXT" lazy-init="true">
            <constructor-arg>
                <bean class="ch.ethz.lthomas.ma.nxt.comm.NXTCommUSB"
                    factory-method="getConnection" lazy-init="true"/>
            </constructor-arg>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean id="Feig" class="ch.ethz.lthomas.ma.feig.FeigJNI"
         lazy-init="true">
            <!-- Com-port to use for connection. -->
            <constructor-arg type="java.lang.String"
             value="/dev/tty.PL2303-004052FD"/>
        </bean>
    </constructor-arg>
    <constructor-arg ref="CoordTranslator"/>
</bean>
```

B.4. Bedienung der Hardware

Ist die Konfiguration abgeschlossen, kann die Hardware vorbereitet und danach das Spiel gestartet werden. Dabei müssen jedoch einige Punkte beachtet werden.

B.4.1. Vorbereitung NXT und Feig Lesegerät

Das LEGO NXT-Modul wird über USB direkt mit dem PC verbunden werden. Das WH40k-GUI findet die entsprechende Schnittstelle automatisch. Es darf jedoch nur ein NXT-Modul mit dem PC verbunden sein. Man muss darauf achten, dass die Batterien im NXT vollständig geladen und das USB-Kabel genügend lang ist, um den beweglichen Teil der LEGO-Konstruktion nicht zu behindern. Das NXT-Modul muss gestartet werden, darf jedoch kein Programm ausführen. Ob die Firmware von LEGO oder diejenige von LEJOS installiert ist, spielt keine Rolle.

Das Lesegerät der Firma Feig kann entweder direkt an einen seriellen Anschluss am PC oder über einen USB-auf-Seriell-Adapter verbunden werden. Dies musste in der Konfiguration bereits festgelegt werden. Das RFID-Lesegerät muss vor dem Starten des Programms in Betrieb sein. Die LED muss grün leuchten und die Antenne angeschlossen sein.

B.4.2. Starten des Spiels

Nun sollte nochmals überprüft werden, ob das NXT-Modul immer noch läuft, da sich dieses nach einiger Zeit selbst ausschalten kann. Ist alles bereit, kann das Programm gestartet werden. Es empfiehlt sich, dieses auf der Kommandozeile oder direkt aus NetBeans zu starten, da mögliche Fehlermeldungen auf der Konsole ausgegeben werden.

B.4.3. Bedienen des Spiels

Das Spielfeld wird wiederholt eingelesen und das virtuelle Feld auf dem Computer aktualisiert. Ein Spiel kann nicht vollständig durchgespielt werden, da einerseits das Spielfeld zu klein und andererseits nach ungefähr 30 Minuten die Batterien leer sind.

Werden diejenigen Figuren bewegt, die über der aktuellen Position der Antenne stehen, so kann es in dieser Runde zu falschen Anzeigen auf dem Bildschirm kommen. Es ist möglich, dass die Figuren nicht an der Start- oder Zielposition, sondern irgendwo dazwischen angezeigt werden. Dies korrigiert sich nach dem nächsten Einlesevorgang automatisch.

Das Spiel darf nicht in dem Moment abgebrochen werden, in dem der Schlitten zurück zur Ursprungsposition fährt. In dieser Situation würde das Modell nicht abbremsen und über den Stopper hinausfahren. Jeder andere Moment kann zum Abbrechen genutzt werden. Dazu muss nur das Programm auf dem Computer beendet werden.

C Konfigurationsdateien

C.1. `configuration.xml`

Diese Datei wird direkt vom Bean-Container des Spring Frameworks geladen. Die wichtigsten Komponenten sind das Initialisieren des Spielfelds und die Koordinatentransformation. Es sind jeweils kommentierte Beispiele für den Betrieb mit Simulator oder der Hardware vorhanden. In A.1 wird die Konfiguration für den Simulator, in B.3 diejenige für die Hardware beschrieben.

C.2. `areas.xml`

Diese Datei enthält die speziellen Terrains, die auf dem Spielfeld vorkommen können. Jedes Terrain wird wir folgt beschrieben:

```
<area type="dangerous" height="1">
    <boundings>
        <bounding>
            <x>200</x><y>200</y>
        </bounding>
        <bounding>
            <x>800</x><y>600</y>
        </bounding>
        <bounding>
            <x>600</x><y>100</y>
        </bounding>
    </boundings>
</area>
```

Die Höhe wird wie in WH40k üblich nur in die Kategorien 1, 2 und 3 unterteilt und hat einen Einfluss beim Berechnen des Sichtkontakts. Der Typ kann einen der folgenden Werte einnehmen:

clear Das Gebiet ist frei von jeglichen Einschränkungen. Dies kann dazu genutzt werden, um nur die Höhe eines Gebiets festzulegen.

difficult Schwieriges Terrain. Es wird ein Difficult-Terrain-Test von den Spielern verlangt.

dangerous Gefährliches Terrain. Es wird ein Dangerous-Terrain-Test verlangt.

impassable Unpassierbares Terrain.

In `boundings` folgt die eigentliche Positionsangabe. Dabei wird für jede Ecke des Bereichs die X- und die Y-Koordinate angegeben. Das Gebiet muss nicht geschlossen werden, dies wird vom Programm automatisch erledigt. Abbildung A.2 (Seite 77) zeigt, was die oben angegebene Definition auf dem Spielfeld erzeugt.

C.3. modelTemplates.xml

Diese Datei enthält Vorlagen für die verschiedenen Typen von Modellen. Fahrzeuge werden dabei anders als einfache Figuren konfiguriert. Das erste Beispiel zeigt ein einfaches Modell:

```
<model name="Termagant" short="Trmg" type="nonvehicle">
    <points>7</points> <!-- Value in combat -->
    <skills>
        <WS>3</WS>  <!-- Weapon Skill -->
        <BS>3</BS>  <!-- Ballistic Skill -->
        <S>3</S>    <!-- Strength -->
        <T>3</T>    <!-- Toughness -->
        <W>1</W>    <!-- Wounds -->
        <I>4</I>    <!-- Initiative -->
        <A>1</A>    <!-- Attacks -->
        <Ld>5</Ld>  <!-- Leadership -->
        <Sv>6+</Sv> <!-- Save -->
    </skills>
    <weapons>
        <weapon>Boltgun</weapon>
    </weapons>
</model>
```

Offensichtlich ist keine Tag-ID angegeben. Diese folgt in der Datei `concreteUnits.xml`. Ebenso fehlt die genaue Definition der Waffe; diese steht in `weaponTemplates.xml`.

Wichtig ist, dass jede Art Einheit einen eindeutigen Namen (Termagant) hat. Der Kurzname (Trmg) wird auf dem virtuellen Spielfeld genutzt, da die langen Namen zuviel Platz benötigen. Die restlichen Attribute sind in der XML-Datei beschrieben und entsprechen direkt den Eigenschaften dieser Einheit.

Die Konfiguration des auf Abbildung A.2 (Seite 77) gezeigten Fahrzeugs ist wesentlich komplexer, da viel mehr Informationen benötigt werden.

```
<model name="Tank␣1" short="Tank1" type="vehicle">
    <points>35</points> <!-- Value in combat -->

    <template>
        <node ref="top">  <x>250</x> <y>0</y>    </node>
        <node ref="left"> <x>-250</x><y>-150</y></node>
        <node ref="right"><x>-250</x><y>150</y> </node>
        <node ref="tlo">  <x>250</x> <y>-130</y></node>
        <node ref="tlu">  <x>230</x> <y>-150</y></node>
        <node ref="tro">  <x>250</x> <y>130</y> </node>
        <node ref="tru">  <x>230</x> <y>150</y> </node>
        <node ref="tru">  <x>230</x> <y>150</y> </node>
        <node ref="wr1">  <x>50</x>  <y>150</y> </node>
        <node ref="wr2">  <x>50</x>  <y>200</y> </node>
        <node ref="wr3">  <x>-25</x> <y>200</y> </node>
        <node ref="wr4">  <x>-50</x> <y>150</y> </node>
        <node ref="wl1">  <x>50</x>  <y>-150</y></node>
        <node ref="wl2">  <x>50</x>  <y>-200</y></node>
        <node ref="wl3">  <x>-25</x> <y>-200</y></node>
        <node ref="wl4">  <x>-50</x> <y>-150</y></node>
        <node ref="wl">   <x>25</x>  <y>-175</y></node>
        <node ref="wr">   <x>25</x>  <y>175</y> </node>
        <node ref="wm">   <x>50</x>  <y>0</y>    </node>
    </template>

    <outline>
        <point ref="tlu"/>  <point ref="tlo"/>
        <point ref="tro"/>  <point ref="tru"/>
        <point ref="wr1"/>  <point ref="wr2"/>
        <point ref="wr3"/>  <point ref="wr4"/>
        <point ref="right"/><point ref="left"/>
        <point ref="wl4"/>  <point ref="wl3"/>
        <point ref="wl2"/>  <point ref="wl1"/>
    </outline>

    <skills>
        <WS>3</WS>  <!-- Weapon Skill -->
        <BS>3</BS>  <!-- Ballistic Skill -->
        <S>3</S>    <!-- Strength -->
        <T>3</T>    <!-- Toughness -->
        <W>1</W>    <!-- Wounds -->
        <I>4</I>    <!-- Initiative -->
        <A>1</A>    <!-- Attacks -->
        <Ld>5</Ld>  <!-- Leadership -->
        <Sv>6+</Sv> <!-- Save -->
    </skills>
    <weapons>
        <weapon ref="wm" angle="0;50">Ion Cannon</weapon>
        <weapon ref="wl" angle="40;40">Ion Cannon</weapon>
        <weapon ref="wr" angle="320;40">Ion Cannon</weapon>
    </weapons>
</model>
```

Die Konfiguration beginnt mit der Aufzählung sämtlicher Punkte, die für dieses Fahrzeug benötigt werden. Jeder Punkt besteht aus einem eindeutigen Namen und den Koordinaten.

Unter `outline` wird eine Liste von bereits definierten Punkten angegeben, die die Aussenlinie des Fahrzeugs festlegen. Die Fähigkeiten (`skills`) der Einheit sind identisch mit denjenigen der einfachen Modelle. Die Waffen besitzen einen Punkt, an dem sie montiert sind, und auch einen Winkel. Der Winkel besteht aus zwei mit einem Strichpunkt getrennten Angaben. Dabei handelt es sich bei der ersten Zahl um den Winkel in Grad und bei der zweiten um eine Abweichung, die angibt, wie weit sich die Waffe auf beide Seiten drehen kann.

Auch für die Fahrzeuge folgen die IDs der Tags erst in der Datei `concreteUnits.xml`. Wichtig ist, dass die Positionen der Tags bereits hier mit einem Namen versehen werden. Später kann nur noch auf den Namen einer Position referenziert werden.

C.4. weaponTemplates.xml

Alle Waffen sämtlicher Figuren werden zentral in dieser Datei definiert. Das Attribut `name` muss jeweils eindeutig sein.

```
<weapon name="Boltgun" type="rapidfire">
    <range>24</range>
    <strength>4</strength>
    <AP>5</AP>      <!-- Armour Piercing Value -->
    <options/>
</weapon>
```

Die in WH40k üblichen Typen von Waffen sind im Spiel verankert und können über `type` referenziert werden. Kommentierte Beispiele zu den Optionen einzelner Waffen befinden sich in obiger Datei.

C.5. concreteUnits.xml

Diese Datei enthält den letzten fehlenden Baustein. Hier werden die einzelnen Einheiten mit allen enthaltenen Modellen aufgelistet. Jedem Modell wird dabei mindestens eine Tag-ID zugewiesen. Die Fahrzeuge benötigen mindestens zwei IDs.

Die Konfiguration für eine Einheit aus zwei Modellen (keine Fahrzeuge) sieht so aus:

```
<unit>
    <owner>1</owner>
    <models>
        <model type="Space␣Marine">
            <tag>0900000081B1946D</tag>
        </model>
        <model type="Space␣Marine">
            <tag>0900000081B1AB1C</tag>
        </model>
    </models>
</unit>
```

Unter `owner` wird die Nummer des Spielers gespeichert, dem diese Einheit gehört. Danach folgt eine Liste aller Modelle dieser Einheit. Für jedes Modell wird mit `type` auf einen Typ aus der Datei `modelTemplates.xml` verwiesen. Mit `tag` wird jedem Modell eine eindeutige Tag-ID zugewiesen.

Fahrzeuge sind grundsätzlich ähnlich aufgebaut, benötigen jedoch mehr als eine Tag-ID pro Modell. Dies wird über die bereits in `modelTemplates.xml` definierten Namen für Punkte gelöst.

```
<unit>
    <owner>1</owner>
    <models>
       <model type="Tank␣1">
           <tag ref="t1">0900000081B1ADB5</tag>
           <tag ref="t2">0900000081B197C3</tag>
        </model>
    </models>
</unit>
```

Für all diese Konfigurationsdateien existiert kein XML-Schema, das die Gültigkeit überprüft. Falsche Werte werden vom Programm schlicht ignoriert. Es empfiehlt sich daher, die vielen vorhandenen Beispiele aus den Dateien den eigenen Bedürfnissen anzupassen und für weitere Einheiten zu duplizieren.

Literaturverzeichnis

[1] Games Workshop. http://www.games-workshop.de/.

[2] Lego NXT Software. http://mindstorms.lego.com/Support/Updates/.

[3] LEJOS. http://lejos.sourceforge.net/.

[4] *LEGO MINDSTORMS NXT Communication Protocol*. LEGO Group, 2006.

[5] *PerGames*, 2007. http://www.pergames.de/.

[6] *Warhammer 40'000 Regelbuch*. Games Workshop, 4 edition, 2008.

[7] B. J. Arnoldus. osx-pl2303: PL2303 USB to Serial Driver for Mac OS X. http://sourceforge.net/projects/osx-pl2303/.

[8] Raffael Bachmann. Erkennen der Position und Ausrichtung von Objekten mit Hilfe von RFID-Technologie. *Semesterarbeit ETH Zürich*, 2008.

[9] Paramvir Bahl and Venkata N. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. Technical report, Microsoft Research, 2000.

[10] Chris Crawford. *The Art of Computer Game Design*. Washington State University Vancouver, 1982.

[11] Andrew Davison. *Killer Game Programming with Java*. O'Reilly Media, Inc., 2005.

[12] Lone Wolf Development. Army Builder. http://www.wolflair.com/index.php?context=army_builder.

[13] Daniel Drake. libusb. http://sourceforge.net/projects/libusb/.

[14] Frank Gutmann and Klaus Rechert. Positionsbestimmung in GSM- und UMTS-Netzwerken. *Semesterarbeit Albert Ludwigs Universität Freiburg*, 2007.

[15] Jeffrey Hightower and Gaetano Borriello. Location sensing techniques. Technical report, University of Washington, 2001.

[16] Jeffrey Hightower, Gaetano Borriello, and Roy Want. SpotON: Indoor location sensing based on RF signal strength. Technical report, University of Washington, 2000.

[17] Steve Hinske. Determining the Position and Orientation of Multi-Tagged Objects Using RFID Technology. *IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 377–381, 2007.

[18] Steve Hinske and Marc Langheinrich. An RFID-based Infrastructure for Automatically Determining the Position and Orientation of Game Objects in Tabletop Games. In *Proceedings of EuroSSC, Zurich*, 2008.

[19] Steve Hinske and Marc Langheinrich. RFIDice - Augmenting Tabletop Dice with RFID. *Journal of Virtual Reality and Broadcasting*, 2008.

[20] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241, New York, NY, USA, 1997. ACM.

[21] Albert Krohn, Tobias Zimmer, and Michael Beigl. Enhancing Tabletop Games with Relative Positioning. In *Advances in Pervasive Computing, Oesterreichische Computer Gesellschaft*, 2004.

[22] Catalyst Game Lab. Classic BattleTech. `http://www.classicbattletech.com/`.

[23] Lionel M. Ni, Yunhao Liu, Yiu Cho Lau, and Abhishek P. Patil. LANDMARC: Indoor Location Sensing Using Active RFID. *Pervasive Computing and Communications*, pages 407–415, 2003.

[24] Johan Peitz, Staffan Björk, and Anu Jäppinen. Wizard's apprentice gameplay-oriented design of a computer-augmented board game. In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, 2006.

[25] E. G. Sarabia, J. R. Llata, J. Arce, and J. P. Oria. Shape Recognition and Orientation Detection for Industrial Applications using Ultrasonic Sensors. In *INTSYS '98: Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*, page 301, 1998.

[26] Thad Starner, Bernt Schiele, and Alex Pentl. Visual contextual awareness in wearable computing. In *In International Symposium on Wearable Computing*, pages 50–57, 1998.

[27] Roy Want. An Introduction to RFID Technology. *IEEE Pervasive Computing*, pages 25–33, 2006.

[28] Wikipedia. Tangible User Interface. `http://en.wikipedia.org/w/index.php?title=Tangible_User_Interface&oldid=239352073`.

[29] Paul Wilson, Daniel Prashanth, and Hamid Aghajan. Utilizing RFID Signaling Scheme for Localization of Stationary Objects and Speed Estimation of Mobile Objects.

[30] Games Workshop. Necromunda Unterwelt. `http://www.games-workshop.de/warhammer40000/specialist/necromunda/index.shtm`.

[31] Games Workshop. Raumflotte GOTHIC. `http://www.games-workshop.de/warhammer40000/specialist/gothic/index.shtm`.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift